

# **WinAPI 32**

## **Crear aplicaciones Windows**

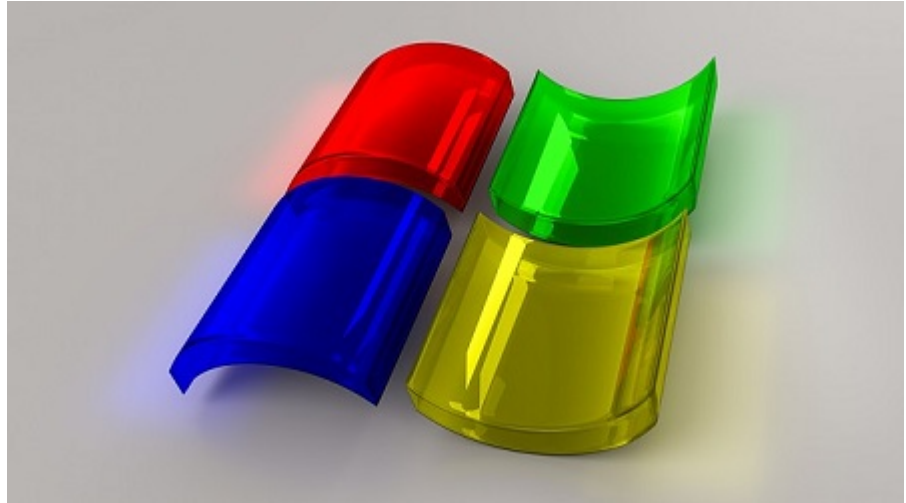


**Salvador Pozo**  
**<http://conclase.net>**

# Introducción

## Requisitos previos

Para el presente curso supondré que estás familiarizado con la programación en C y C++ y también con las aplicaciones y el entorno Windows, al



Logo de Windows

menos al nivel de usuario. Sin embargo, no se requerirán muchos más conocimientos.

El curso pretende ser una explicación de la forma en que se realizan los programas en Windows usando el API. Las explicaciones de las funciones y los mensajes del API son meras traducciones del fichero de ayuda de WIN32 de Microsoft, y sólo se incluyen como complemento.

Para empezar, vamos a ponernos en antecedentes. Veamos primero algunas características especiales de la programación en Windows.

## Independencia de la máquina

Los programas Windows son independientes de la máquina en la que se ejecutan (o al menos deberían serlo), el acceso a los

dispositivos físicos se hace a través de interfaces, y nunca se accede directamente a ellos. Esta es una de las principales ventajas para el programador, ya que no hay que preocuparse por el modelo de tarjeta gráfica o de impresora, la aplicación funcionará con todas, y será el sistema operativo el que se encargue de que así sea.

## Recursos

Un concepto importante es el de recurso. Desde el punto de vista de Windows, un recurso es todo aquello que puede ser usado por una o varias aplicaciones. Existen recursos físicos, que son los dispositivos que componen el ordenador, como la memoria, la impresora, el teclado o el ratón y recursos virtuales o lógicos, como los gráficos, los iconos o las cadenas de caracteres.

Por ejemplo, si nuestra aplicación requiere el uso de un puerto serie, primero debe averiguar si está disponible, es decir, si existe y si no lo está usando otra aplicación; y después lo reservará para su uso. Esto es necesario porque este tipo de recurso no puede ser compartido.

Lo mismo pasa con la memoria o con la tarjeta de sonido, aunque son casos diferentes. Por ejemplo, la memoria puede ser compartida, pero de una forma general, cada porción de memoria no puede compartirse, (al menos en los casos normales, veremos que es posible hacer aplicaciones con memoria compartida), y se trata de un recurso finito. Las tarjetas de sonido, dependiendo del modelo, podrán o no compartirse por varias aplicaciones. Otros recursos como el ratón y el teclado también se comparten, pero se asigna su uso automáticamente a la aplicación activa, a la que normalmente nos referiremos como la que tiene el "foco", es decir, la que mantiene contacto con el usuario.

Desde nuestro punto de vista, como programadores, también consideramos recursos varios componentes como los menús, los iconos, los cuadros de diálogo, las cadenas de caracteres, los mapas de bits, los cursores, etc. En sus programas, el Windows

almacena separados el código y los recursos, dentro del mismo fichero, y estos últimos pueden ser editados por separado, permitiendo por ejemplo, hacer versiones de los programas en distintos idiomas sin tener acceso a los ficheros fuente de la aplicación.

## **Ventanas**

La forma en que se presentan las aplicaciones Windows (al menos las interactivas) ante el usuario, es la ventana. Supongo que todos sabemos qué es una ventana: un área rectangular de la pantalla que se usa de interfaz entre la aplicación y el usuario.

Cada aplicación tiene al menos una ventana, la ventana principal, y todas las comunicaciones entre usuario y aplicación se canalizan a través de una ventana. Cada ventana comparte el espacio de la pantalla con otras ventanas, incluso de otras aplicaciones, aunque sólo una puede estar activa, es decir, sólo una puede recibir información del usuario.

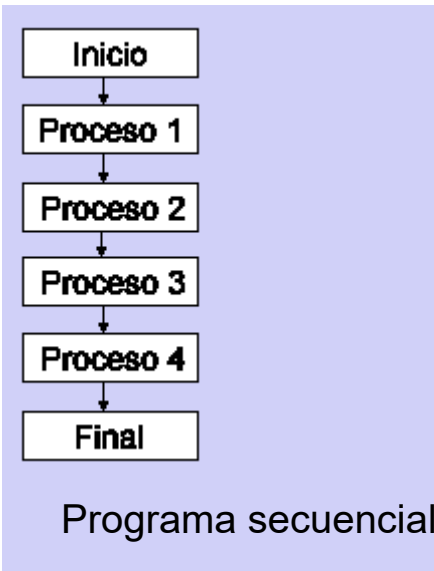
## **Eventos**

Los programas en Windows están orientados a eventos, esto significa que normalmente los programas están esperando a que se produzca un acontecimiento que les incumba, y mientras tanto permanecen aletargados o dormidos.

Un evento puede ser por ejemplo, el movimiento del ratón, la activación de un menú, la llegada de información desde el puerto serie, una pulsación de una tecla...

Esto es así porque Windows es un sistema operativo multitarea, y el tiempo del microprocesador ha de repartirse entre todos los programas que se estén ejecutando. Si los programas fueran secuenciales puros, esto no sería posible, ya que hasta que una aplicación finalizara, el sistema no podría atender al resto.

Estructura de programa secuencial:



Estructura de programa por eventos:

## Proyectos

Debido a la complejidad



de los programas Windows, normalmente los dividiremos en varios ficheros fuente, que compilaremos por separado y enlazaremos juntos.

Cada compilador puede tener diferencias, más o menos grandes, a la hora de trabajar con proyectos. Sin embargo creo que no deberías tener grandes dificultades para adaptarte a cada uno de ellos.

En el presente curso trabajaremos con el compilador de "Bloodshed", que es público y gratuito, y puede descargarse de Internet en la siguiente URL: <http://www.bloodshed.net/>.

Para crear un proyecto Windows usando este compilador elegiremos el menú "File/New Project". Se abrirá un cuadro de diálogo donde podremos elegir el tipo de proyecto. Elegiremos "Windows Application" y "C++ Project". A continuación pulsamos "Aceptar".

El compilador crea un proyecto con un fichero C++, con el esqueleto de una aplicación para una ventana, a partir de ahí empieza nuestro trabajo.

## Convenciones

En parte para que no te resulte muy difícil adaptarte a la terminología de Windows, y a la documentación existente, y en parte para seguir mi propia costumbre, en la mayoría de los casos me referiré a componentes y propiedades de Windows con sus nombres en inglés. Por ejemplo, hablaremos de "button", "check box", "radio button", "list box", "combo box" o "property sheet", aunque algunas veces traduzca sus nombre a español, por ejemplo, "dialog box" se nombrará a menudo como "cuadro de diálogo".

Además hablaremos a menudo de ventanas "overlapped" o superponibles, que son las ventanas corrientes. Para el término "pop-up" he desistido de buscar una traducción.

También se usaran a menudo, con relación a "check boxes", términos ingleses como checked, unchecked o grayed, en lugar de marcado, no marcado o gris.

Owner-draw, es un estilo que indica que una ventana o control no es la encargada de actualizarse en pantalla, esa responsabilidad es transferida a la ventana dueña del control o ventana.

Para "bitmap" se usará a menudo la expresión "mapa de bits".

## Controles

Los controles son la forma en que las aplicaciones Windows intercambian datos con el usuario. Normalmente se usan dentro de los cuadros de diálogo, pero en realidad pueden usarse en cualquier ventana.

Existen bastantes, y los iremos viendo poco a poco, al mismo tiempo que aprendemos a manejarlos.

Los más importantes y conocidos son:

- control static: son etiquetas, marcos, iconos o dibujos.
- control edit: permiten que el usuario introduzca datos alfanuméricos en la aplicación.
- control list box: el usuario puede escoger entre varias opciones de una lista.

- control combo box: es una combinación entre un edit y un list box.
- control scroll bar: barras de desplazamiento, para la introducción de valores entre márgenes definidos.
- control button: realizan acciones o comandos, de button de derivan otros dos controles muy comunes:
  - control check box: permite leer variables de dos estados "checked" o "unchecked"
  - control radio button: se usa en grupos, dentro de cada grupo sólo uno puede ser activado.

# Capítulo 1 Componentes de una ventana

Veamos ahora los elementos que componen una ventana, aunque más adelante veremos que no todos tienen por qué estar presentes en todas las ventanas.

## El borde de la ventana

Hay varios tipos, dependiendo de que estén o no activas las opciones de cambiar el tamaño de la ventana.

Se trata de un área estrecha alrededor de la ventana que permite cambiar su tamaño (1).



## Barra de título

Zona en la parte superior de la ventana que contiene el icono y el título de la ventana, esta zona también se usa para mover la ventana a través de la pantalla, y mediante doble clic, para cambiar entre el modo maximizado y tamaño normal (2).

## Caja de minimizar

Pequeña área cuadrada situada en la parte derecha de la barra de título que sirve para disminuir el tamaño de la ventana. Antes de



la aparición del Windows 95 la ventana se convertía a su forma icónica, pero desde la aparición del Windows 95 los iconos desaparecieron, la ventana se oculta y sólo permanece un botón en la barra de estado (3).

## **Caja de maximizar**

Pequeña área cuadrada situada en la parte derecha de la barra de título que sirve para agrandar la ventana para que ocupe toda la pantalla. Cuando la ventana está maximizada, se sustituye por la caja de restaurar (4).

## **Caja de cerrar**

Pequeña área cuadrada situada en la parte derecha de la barra de título que sirve para cerrar la ventana. (5)

## **Caja de control de menú**

Pequeña área cuadrada situada en la parte izquierda de la barra de título, normalmente contiene el icono de la ventana, y sirve para desplegar el menú del sistema (6).

## **Menú**

O menú del sistema. Se trata de una ventana especial que contiene las funciones comunes a todas las ventanas, también accesibles desde las cajas y el borde, como minimizar, restaurar, maximizar, mover, cambiar tamaño y cerrar. Este menú se despliega al pulsar sobre la caja de control de menú.

## **Barra de menú**

Zona situada debajo de la barra de título, contiene los menús de la aplicación (7).

## **Barra de scroll horizontal**

Barra situada en la parte inferior de la ventana, permite desplazar horizontalmente la vista del área de cliente (8).

## **Barra de scroll vertical**

Barra situada en la parte derecha de la ventana, permite desplazar verticalmente la vista del área de cliente (9).

## **El área de cliente**

Es la zona donde el programador sitúa los controles, y los datos para el usuario. En general es toda la superficie de la ventana lo que no está ocupada por las zonas anteriores (10).

# Capítulo 2 Notación Húngara

La notación húngara es un sistema usado normalmente para crear los nombres de variables, tipos y estructuras cuando se programa en Windows. Es el sistema usado en la programación del sistema operativo, y también por la mayoría de los programadores. A veces también usaremos este sistema en algunos ejemplos de este curso, pero sobre todo, nos ayudará a interpretar el tipo básico al que pertenece cada estructura, miembro, o tipo definido.

Consiste en prefijos en minúsculas que se añaden a los nombres de las variables, y que indican su tipo; en el caso de tipos definidos, las letras del prefijo estarán en mayúscula. El resto del nombre indica, lo más claramente posible, la función que realiza la variable o tipo.

## Prefijo Significado

b	Booleano
c	Carácter (un byte)
dw	Entero largo de 32 bits sin signo (DOBLE WORD)
f	Flags empaquetados en un entero de 16 bits
h	Manipulador de 16 bits (HANDLE)
l	Entero largo de 32 bits
lp	Puntero a entero largo de 32 bits
lpfn	Puntero largo a una función que devuelve un entero
lpsz	Puntero largo a una cadena terminada con cero
n	Entero de 16 bits
p	Puntero a entero de 16 bits
pt	Coordenadas (x, y) empaquetadas en un entero de 32 bits
rgb	Valor de color RGB empaquetado en un entero de 32 bits
sz	Cadena terminada en cero
u	Sin signo (unsigned)

w      Entero corto de 16 bits sin signo (WORD)

## Ejemplos

nContador: la variable es un entero que se usará como contador.

szNombre: una cadena terminada con cero que almacena un nombre.

bRespuesta: una variable booleana que almacena una respuesta.

Ejemplos de tipos definidos por el API de Windows:

UINT: entero sin signo. Windows redefine los enteros para asegurar que el tamaño en bits es siempre el mismo para todas las variables del API.

LRESULT: entero largo usado como valor de retorno.

WPARAM: entero corto de 16 bits usado como parámetro.

LPARAM: entero largo de 32 bits usado como parámetro.

LPSTR: puntero largo a una cadena de caracteres. En el API de 32 bits no existen distinciones entre punteros largos y cortos, pero la nomenclatura se conserva por compatibilidad.

LPCREATESTRUCT: puntero a una estructura  
[CREATESTRUCT](#).

# Capítulo 3 Estructura de un programa Windows GUI

Hay algunas diferencias entre la estructura de un programa C/C++ normal, y la correspondiente a un programa Windows GUI. Algunas de estas diferencias se deben a que los programas GUI están basados en mensajes, otros son sencillamente debidos a que siempre hay un determinado número de tareas que hay que realizar.

```
// Ficheros include:
#include <windows.h>

// Prototipos:
LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM,
LPARAM);

// Función de entrada:
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpszCmdParam,
                   int nCmdShow)
{
    // Declaración:
    // Inicialización:
    // Bucle de mensajes:
    return Message.WParam;
}

// Definición de funciones:
```

## Cabeceras

Lo primero es lo primero, para poder usar las funciones del API de Windows hay que incluir al menos un fichero de cabecera, pero generalmente no bastará con uno.

El fichero <windows.h> lo que hace es incluir la mayoría de los ficheros de cabecera corrientes en aplicaciones GUI, pero podemos incluir sólo los que necesitemos, siempre que sepamos cuales son. Por ejemplo, la función [WinMain](#) está declarada en el fichero de cabecera *winbase.h*.

Generalmente esto resultará incómodo, ya que para cada nueva función, mensaje o estructura tendremos que comprobar, y si es necesario, incluir nuevos ficheros. Es mejor usar *windows.h* directamente.

## Prototipos

Cada tipo (o clase) de ventana que usemos en nuestro programa (normalmente sólo será una), o cada cuadro de diálogo (de estos puede haber muchos), necesitará un procedimiento propio, que deberemos declarar y definir. Siguiendo la estructura de un programa C, esta es la zona normal de declaración de prototipos.

## Función de entrada, WinMain

La función de entrada de un programa Windows es "[WinMain](#)", en lugar de la conocida "main". Normalmente, la definición de esta función cambia muy poco de una aplicaciones a otras. Se divide en tres partes claramente diferenciadas: declaración, inicialización y bucle de mensajes.

### Parámetros de entrada de "WinMain"

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
LPSTR lpszCmdParam, int nCmdShow)
```

La función WinMain tiene cuatro parámetros de entrada:

- `hInstance` es un manipulador para la instancia del programa que estamos ejecutando. Cada vez que se ejecuta una aplicación, Windows crea una Instancia para ella, y le pasa un manipulador de dicha instancia a la aplicación.
- `hPrevInstance` es un manipulador a instancias previas de la misma aplicación. Como Windows es multitarea, pueden existir varias versiones de la misma aplicación ejecutándose, varias instancias. En Windows 3.1, este parámetro nos servía para saber si nuestra aplicación ya se estaba ejecutando, y de ese modo se podían compartir los datos comunes a todas las instancias. Pero eso era antes, ya que en Win32 usa un segmento distinto para cada instancia y este parámetro es siempre **NULL**, sólo se conserva por motivos de compatibilidad.
- `lpCmdParam`, esta cadena contiene los argumentos de entrada del comando de línea.
- `nCmdShow`, este parámetro especifica cómo se mostrará la ventana. Para ver sus posibles valores consultar [valores de `nCmdShow`](#). Se recomienda no usar este parámetro en la función [ShowWindow](#) la primera vez que se ésta es llamada. En su lugar debe usarse el valor **SW\_SHOWDEFAULT**.

## Función WinMain típica

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
    LPSTR lpCmdParam, int nCmdShow)
{
    /* Declaración: */
    HWND hwnd;
    MSG mensaje;
    WNDCLASSEX wincl;

    /* Inicialización: */
    /* Estructura de la ventana */
    wincl.hInstance = hInstance;
    wincl.lpszClassName = "NUESTRA_CLASE";
    wincl.lpfnWndProc = WindowProcedure;
    wincl.style = CS_DBLCLKS;
    wincl.cbSize = sizeof(WNDCLASSEX);
```

```

/* Usar icono y puntero por defecto */
wincl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wincl.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
wincl.hCursor = LoadCursor(NULL, IDC_ARROW);
wincl.lpszMenuName = NULL;
wincl.cbClsExtra = 0;
wincl.cbWndExtra = 0;
wincl.hbrBackground = (HBRUSH)COLOR_BACKGROUND;

/* Registrar la clase de ventana, si falla, salir del
programa */
if(!RegisterClassEx(&wincl)) return 0;

hwnd = CreateWindowEx(
    0,
    "NUESTRA_CLASE",
    "Ejemplo 001",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    544,
    375,
    HWND_DESKTOP,
    NULL,
    hThisInstance,
    NULL
);

ShowWindow(hwnd, SW_SHOWDEFAULT);

/* Bucle de mensajes: */
while(TRUE == GetMessage(&mensaje, 0, 0, 0))
{
    TranslateMessage(&mensaje);
    DispatchMessage(&mensaje);
}

return mensaje.wParam;
}

```

## Declaración

En la primera zona declararemos las variables que necesitamos para nuestra función [WinMain](#), que como mínimo serán tres:



- **HWND** hWnd, un manipulador para la ventana principal de la aplicación. Ya sabemos que nuestra aplicación necesitará al menos una ventana.
- **MSG** Message, una variable para manipular los mensajes que lleguen a nuestra aplicación.
- **WNDCLASSEX** wincl, una estructura que se usará para registrar la clase particular de ventana que usaremos en nuestra aplicación. Existe otra estructura para registrar clases que se usaba antiguamente, pero que ha sido desplazada por esta nueva versión, se trata de **WNDCLASS**.

## Inicialización

Esta zona se encarga de registrar la clase o clases de ventana, crear la ventana y visualizarla en pantalla.

Para registrar la clase primero hay que rellenar adecuadamente la estructura **WNDCLASSEX**, que define algunas características que serán comunes a todas las ventanas de una misma clase, como color de fondo, icono, menú por defecto, el procedimiento de ventana, etc. Después hay que llamar a la función **RegisterClassEx**.

En el caso de usar una estructura **WNDCLASS** se debe registrar la clase usando la función **RegisterClass**.

A continuación se crea la ventana usando la función **CreateWindowEx**, la función **CreateWindow** ha caído prácticamente en desuso. Cualquiera de estas dos funciones nos devuelve un manipulador de ventana que podemos necesitar en otras funciones, sin ir más lejos, la siguiente.

Pero esto no muestra la ventana en la pantalla. Para que la ventana sea visible hay que llamar a la función **ShowWindow**. La primera vez que se llama a ésta función, después de crear la ventana, se puede usar el parámetro **nCmdShow** de **WinMain** como parámetro o mejor aún, como se recomienda por Windows, el valor **SW\_SHOWDEFAULT**.

## Bucle de mensajes

Este es el núcleo de la aplicación, como se ve en el ejemplo el programa permanece en este bucle mientras la función `GetMessage` retorne con un valor `TRUE`.

```
while(TRUE == GetMessage(&mensaje, 0, 0, 0)) {  
    TranslateMessage(&mensaje);  
    DispatchMessage(&mensaje);  
}
```

Este es el bucle de mensajes recomendable, aunque no sea el que se usa habitualmente. La razón es que la función `GetMessage` puede retornar tres valores: `TRUE`, `FALSE` ó `-1`. El valor `-1` indica un error, así que en este caso se debería abandonar el bucle.

El bucle de mensajes que encontraremos habitualmente es este:

```
while(GetMessage(&mensaje, 0, 0, 0)) {  
    TranslateMessage(&mensaje);  
    DispatchMessage(&mensaje);  
}
```

**Nota:**

El problema con este bucle es que si `GetMessage` regresa con un valor `-1`, que indica un error, la condición del "while" se considera verdadera, y el bucle continúa. Si el error es permanente, el programa jamás terminará.

La función `TranslateMessage` se usa para traducir los mensajes de teclas virtuales a mensajes de carácter. Veremos esto con más detalle en el capítulo dedicado al teclado (cap. 34).

Los mensajes traducidos se reenvían a la lista de mensajes del proceso, y se recuperarán con las siguientes llamadas a `GetMessage`.

La función `DispatchMessage` envía el mensaje al procedimiento de ventana, donde será tratado adecuadamente. El próximo capítulo está dedicado al procedimiento de ventana, y al final de él estaremos en disposición de crear nuestro primer programa Windows.

## Definición de funciones

En esta parte definiremos, entre otras cosas, los procedimientos de ventana, que se encargan de procesar los mensajes que lleguen a cada ventana.

# Capítulo 4 El procedimiento de ventana

Cada ventana tiene una función asociada, esta función se conoce como procedimiento de ventana, y es la encargada de procesar adecuadamente todos los mensajes enviados a una determinada clase de ventana. Es la responsable de todo lo relativo al aspecto y al comportamiento de una ventana.

Normalmente, estas funciones están basadas en una estructura "switch" donde cada "case" corresponde a un determinado tipo de mensaje.

## Sintaxis

```
LRESULT CALLBACK WindowProcedure(  
    HWND hwnd,      // Manipulador de ventana  
    UINT msg,        // Mensaje  
    WPARAM wParam,  // Parámetro palabra, varía  
    LPARAM lParam    // Parámetro doble palabra, varía  
);
```

- hwnd es el manipulador de la ventana a la que está destinado el mensaje.
- msg es el código del mensaje.
- wParam es el parámetro de tipo palabra asociado al mensaje.
- lParam es el parámetro de tipo doble palabra asociado al mensaje.

Podemos considerar este prototipo como una *plantilla* para crear nuestros propios procedimientos de ventana. El nombre de la función puede cambiar, pero el valor de retorno y los parámetros

deben ser los mismos. El miembro *lpfnWndProc* de la estructura **WNDCLASS** es un puntero a una función de este tipo, esa función es la que se encargará de procesar todos los mensajes para esa clase de ventana. Cuando registremos nuestra clase de ventana, tendremos que asignar a ese miembro el puntero a nuestro procedimiento de ventana.

Para más detalles sobre la función de procedimiento de ventana, consultar [WindowProc](#).

## Prototipo de procedimiento de ventana

```
LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM,
LPARAM);
```

## Implementación de procedimiento de ventana simple

```
/* Esta función es llamada por la función del API
DispatchMessage() */
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT msg, WPARAM
wParam, LPARAM lParam)
{
    switch (msg)                                /* manipulador del mensaje
*/
    {
        case WM_DESTROY:
            PostQuitMessage(0);                /* envía un mensaje
WM_QUIT a la cola de mensajes */
            break;
        default:                                /* para los mensajes de
los que no nos ocupamos */
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

En general, habrá tantos procedimientos de ventana como programas diferentes y todos serán distintos, pero también tendrán algo en común: todos ellos procesarán los mensajes que lleguen a una clase de ventana.

En este ejemplo sólo procesamos un tipo de mensaje, se trata de [WM\\_DESTROY](#) que es el mensaje que se envía a una ventana cuando se recibe un comando de cerrar, ya sea por menú o mediante el icono de aspa en la esquina superior derecha de la ventana.

Este mensaje sólo sirve para informar a la aplicación de que el usuario tiene la intención de abandonar la aplicación, y le da una oportunidad de dejar las cosas en su sitio: cerrar ficheros, liberar memoria, guardar variables, etc. Incluso, la aplicación puede decidir que aún no es el momento adecuado para abandonar la aplicación. En el caso del ejemplo, efectivamente cierra la aplicación, y lo hace enviándole un mensaje [WM\\_QUIT](#), mediante la función [PostQuitMessage](#).

El resto de los mensajes se procesan en el caso "default", y simplemente se cede su tratamiento a la función del API que hace el proceso por defecto para cada mensaje, [DefWindowProc](#).

Este es el camino que sigue el mensaje [WM\\_QUIT](#) cuando llega, ya que el proceso por defecto para este mensaje es cerrar la aplicación.

En posteriores capítulos veremos como se complica paulatinamente esta función, añadiendo más y más mensajes.

## Primer ejemplo de programa Windows

Ya estamos en condiciones de crear nuestro primer programa Windows, que sólo mostrará una ventana en pantalla.

# Capítulo 5 Menús 1

Ahora que ya sabemos hacer el esqueleto de una aplicación Windows, veamos el primer medio para comunicarnos con ella.

Supongo que todos sabemos lo que es un menú: se trata de una ventana un tanto especial, del tipo pop-up, que contiene una lista de comandos u opciones entre las cuales el usuario puede elegir.

Cuando se usan en una aplicación, normalmente se agrupan varios menús bajo una barra horizontal, (que no es otra cosa que un menú), dividida en varias zonas o ítems.

Cada ítem de un menú, (salvo los separadores y aquellos que despliegan nuevos menús), tiene asociado un identificador. El valor de ese identificador se usará por la aplicación para saber qué opción se activó por el usuario, y decidir las acciones a tomar en consecuencia.

Existen varias formas de añadir un menú a una ventana, veremos cada una de ellas por separado.

También es posible desactivar o inhibir algunas opciones para que no estén disponibles para el usuario.

## Usando las funciones para inserción ítem a ítem

Este es el sistema más rudimentario, pero como ya veremos en el futuro, en ocasiones puede ser muy útil. Empezaremos viendo este sistema porque ilustra mucho mejor la estructura de los menús.

Tomemos el ejemplo del capítulo anterior y definamos algunas constantes:

```
#define CM_PRUEBA 100
```

```
#define CM_SALIR 101
```

Y añadamos la declaración de una función en la zona de prototipos:

```
void InsertarMenu(HWND);
```

Al final del programa añadimos la definición de esta función:

```
void InsertarMenu(HWND hWnd)
{
    HMENU hMenu1, hMenu2;

    hMenu1 = CreateMenu(); /* Manipulador de la barra de menú */
    hMenu2 = CreateMenu(); /* Manipulador para el primer menú pop-up */
    AppendMenu(hMenu2, MF_STRING, CM_PRUEBA, "&Prueba"); /* 1º ítem */
    AppendMenu(hMenu2, MF_SEPARATOR, 0, NULL); /* 2º ítem (separador) */
    AppendMenu(hMenu2, MF_STRING, CM_SALIR, "&Salir"); /* 3º ítem */
    /* Inserción del menú pop-up */
    AppendMenu(hMenu1, MF_STRING | MF_POPUP, (UINT)hMenu2, "&Principal");
    SetMenu(hWnd, hMenu1); /* Asigna el menú a la ventana hWnd */
}
```

Y por último, sólo nos queda llamar a nuestra función, insertaremos ésta llamada justo antes de visualizar la ventana.

```
...
    InsertarMenu(hWnd);
    ShowWindow(hWnd, SW_SHOWDEFAULT);
...
```



Veamos cómo funciona "InsertarMenu".

La primera novedad son las variables del tipo `HMENU`. `HMENU` es un tipo de manipulador especial para menús. Necesitamos dos variables de este tipo, una para manipular la barra de menú, `hMenu1`. La otra para manipular cada uno de los menús pop-up, en este caso sólo uno, `hMenu2`.

De momento haremos una barra de menú con un único elemento que será un menú pop-up. Después veremos como implementar menús más complejos.

Para crear un menú usaremos la función `CreateMenu`, que crea un menú vacío.

Para ir añadiendo ítems a cada menú usaremos la función `AppendMenu`. Esta función tiene varios argumentos:

El primero es el menú donde queremos insertar el nuevo ítem.

El segundo son las opciones o atributos del nuevo ítem, por ejemplo `MF_STRING`, indica que se trata de un ítem de tipo texto, `MF_SEPARATOR`, es un ítem separador y `MF_POPUP`, indica que se trata de un menú que desplegará un nuevo menú pop-up.

El siguiente parámetro puede tener distintos significados:

- Puede ser un identificador de comando, este identificador se usará para comunicar a la aplicación si el usuario seleccionó un determinado ítem.
- Un manipulador de menú, si el ítem tiene el flag `MF_POPUP`, en este caso hay que hacer un casting a (`UINT`).
- O también puede ser cero, si se trata de un separador.

El último parámetro es el texto del ítem, cuando se ha especificado el flag `MF_STRING`, más adelante veremos que los ítems pueden ser también bitmaps. Normalmente se trata de una cadena de texto. Pero hay una peculiaridad interesante, para indicar la tecla que activa un determinado ítem de un menú se muestra la letra correspondiente subrayada. Esto se consigue insertando un '&' justo antes de la letra que se quiere usar como atajo, por ejemplo, en el ítem "&Prueba" esta letra será la 'P'.

Por último [SetMenu](#), asigna un menú a una ventana determinada. El primer parámetro es el manipulador de la ventana, y el segundo el del menú.

Prueba estas funciones y juega un rato con ellas. A continuación veremos cómo hacer que nuestra aplicación responda a los mensajes del menú.

## Uso básico de MessageBox

Antes de aprender a visualizar texto en la ventana, usaremos un mecanismo más simple para informar al usuario de cualquier cosa que pase en nuestra aplicación. Este mecanismo no es otro que el cuadro de mensaje (message box), que consiste en una pequeña ventana con un mensaje para el usuario y uno o varios botones, según el tipo de cuadro de mensaje que usemos. En nuestros primeros ejemplos, el cuadro de mensaje sólo incluirá el botón de "Aceptar".

Para visualizar un cuadro de mensaje simple, usaremos la función [MessageBox](#). En nuestros ejemplos bastará con la siguiente forma:

```
MessageBox(hWnd, "Texto de mensaje", "Texto de título",  
MB_OK) ;
```

Esto mostrará un pequeño cuadro de diálogo con el texto y el título especificados y un botón de "Aceptar". El cuadro se cerrará al pulsar el botón o al pulsar la tecla de Retorno.

## Respondiendo a los mensajes del menú

Las activaciones de los menús se reciben mediante un mensaje [WM\\_COMMAND](#).

Para procesar estos mensajes, si sólo podemos recibir mensajes desde un menú, únicamente nos interesa la palabra de menor peso del parámetro wParam del mensaje.

Modifiquemos el procedimiento de ventana para procesar los mensajes de nuestro menú:

```
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT msg, WPARAM
wParam, LPARAM lParam)
{
    switch (msg)                                /* manipulador del mensaje
*/
    {
        case WM_COMMAND:
            switch(LOWORD(wParam)) {
                case CM_PRUEBA:
                    MessageBox(hwnd, "Comando: Prueba",
"Mensaje de menú", MB_OK);
                    break;
                case CM_SALIR:
                    MessageBox(hwnd, "Comando: Salir", "Mensaje
de menú", MB_OK);
                    /* envía un mensaje WM_QUIT a la cola de
mensajes */
                    PostQuitMessage(0);
                    break;
            }
            break;
        case WM_DESTROY:
            /* envía un mensaje WM_QUIT a la cola de mensajes
*/
            PostQuitMessage(0);
            break;
        default: /* para los mensajes de los que no nos
ocupamos */
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

Sencillo, ¿no?.

Observa que hemos usado la macro **LOWORD** para extraer el identificador del ítem del parámetro wParam. Después de eso, todo

es más fácil.

También se puede ver que hemos usado la misma función para salir de la aplicación que para el mensaje `WM_DESTROY`: la función `PostQuitMessage`.

## Ejemplo 2

Este ejemplo contiene todo lo que hemos visto sobre los menús hasta ahora.

## Ficheros de recursos

Veamos ahora una forma más sencilla y más frecuente de implementar menús.

Lo normal es implementar los menús desde un fichero de recursos, el sistema que hemos visto sólo se usa en algunas ocasiones, para crear o modificar menús durante la ejecución de la aplicación.

Es importante adquirir algunas buenas costumbres cuando se trabaja con ficheros de recursos.

1. Usaremos siempre etiquetas como identificadores para los ítems de los menús, y nunca valores numéricos literales.
2. Crearemos un fichero de cabecera con las definiciones de los identificadores, en nuestro ejemplo se llamará "win003.h".
3. Incluiremos este fichero de cabecera tanto en el fichero de recursos y como en el del código fuente de nuestra aplicación.

Partimos de un proyecto nuevo: win003. Pero usaremos el código modificado del ejemplo1.

Para ello creamos un nuevo proyecto de tipo GUI, al que llamaremos Win003, y copiamos el contenido de "ejemplo1.c" en un fichero vacío al que nombraremos como "win003.c".

A continuación crearemos el fichero de identificadores.

Añadimos el fichero de cabecera a nuestro proyecto. Desde Code::Blocks ésto se hace pulsando con el botón derecho del ratón "Archivo nuevo" o el menú de "Archivo->nuevo", nos preguntará si queremos añadir el archivo al proyecto, a lo que contestaremos que sí, y le pondremos un nombre, en este caso "win003.h".

Introducimos en é los identificadores:

```
#define CM_PRUEBA 100
#define CM_SALIR 101
```

En el fichero "win003.c" añadimos la línea:

```
#include "win003.h"
```

Justo después de la línea "#include <windows.h>".

Ahora añadiremos el fichero de recursos. Para ello haremos lo mismo que hemos hecho con el fichero "ids.h", pero usaremos el nombre "win003.rc".

En la primera línea introducimos la siguiente línea:

```
#include "win003.h"
```

Y a continuación escribimos:

```
Menu MENU
BEGIN
    POPUP "&Principal"
        BEGIN
            MENUITEM "&Prueba", CM_PRUEBA
            MENUITEM SEPARATOR
            MENUITEM "&Salir", CM_SALIR
        END
    END
END
```

---

En un fichero de recursos podemos crear toda la estructura de un menú fácilmente. Este ejemplo crea una barra de menú con una columna "Principal", con dos opciones: "Prueba" y "Salir", y con un separador entre ellas.

La sintaxis es sencilla, definimos el menú mediante una cadena identificadora, sin comillas, seguida de la palabra **MENU**. Entre las palabras **BEGIN** y **END** podemos incluir ítems, separadores u otras columnas. Para incluir columnas usamos una sentencia del tipo **POPUP** seguida de la cadena que se mostrará como texto en el menú. Cada **POPUP** se comporta del mismo modo que un **MENU**.

Los ítems se crean usando la palabra **MENUITEM** seguida de la cadena que se mostrará en el menú, una coma, y el comando asignado a ese ítem, que puede ser un número entero, o, como en este caso, una macro definida.

Los separadores se crean usando **MENUITEM** seguido de la palabra **SEPARATOR**.

Observarás que las cadenas que se muestran en el menú contienen un símbolo & en su interior, por ejemplo "&Prueba". Este símbolo indica que la siguiente letra puede usarse para activar la opción del menú desde el teclado, usando la tecla [ALT] más la letra que sigue al símbolo &. Para indicar eso, en pantalla, esa letra se muestra subrayada, en este ejemplo "Prueba".

Ya podemos cerrar el cuadro de edición del fichero de recursos.

Para ver más detalles sobre el uso de este recurso puedes consultar las claves: **MENU**, **POPUP** y **MENUITEM**.

## Cómo usar los recursos de menú

Ahora tenemos varias opciones para usar el menú que acabamos de crear.

Primero veremos cómo cargarlo y asignarlo a nuestra ventana, ésta es la forma que más se parece a la del ejemplo del capítulo

anterior. Para ello basta con insertar este código antes de llamar a la función [ShowWindow](#):

```
HMENU hMenu;  
...  
hMenu = LoadMenu(hInstance, "Menu");  
SetMenu(hWnd, hMenu);
```

O simplemente:

```
SetMenu(hWnd, LoadMenu(hInstance, "Menu"));
```

La función [LoadMenu](#) se encarga de cargar el recurso de menú, para ello hay que proporcionarle un manipulador de la instancia a la que pertenece el recurso y el nombre del menú.

Otro sistema, más sencillo todavía, es asignarlo como menú por defecto de la clase. Para esto basta con la siguiente asignación:

```
WNDCLASSEX wincl;  
...  
wincl.lpszMenuName = "Menu";
```

Y por último, también podemos asignar un menú cuando creamos la ventana, especificándolo en la llamada a [CreateWindowEx](#):

```
hwnd = CreateWindowEx(  
    0,  
    "NUESTRA_CLASE",  
    "Ejemplo 003",  
    WS_OVERLAPPEDWINDOW,  
    CW_USEDEFAULT,  
    CW_USEDEFAULT,  
    544,
```

```
        375,  
        HWND_DESKTOP,  
        LoadMenu(hInstance, "Menu"), /* Carga y  
asignación de menú */  
        hInstance,  
        NULL  
    );
```

El tratamiento de los comandos procedentes del menú es igual que en el apartado anterior.

## Ejemplo 3



# Capítulo 6 Diálogo básico

Los cuadros de diálogo son la forma de ventana más habitual de comunicación entre una aplicación Windows y el usuario. Para facilitar la tarea del usuario a la hora de introducir datos, existen varios tipos de controles, cada uno de ellos diseñado para un tipo específico de información. Los más comunes son los "static", "edit", "button", "listbox", "scroll", "combobox", "group", "checkboxbutton" y "radiobutton". A partir de Windows 95 se introdujeron varios controles nuevos: "updown", "listview", "treeview", "gauge", "tab" y "trackbar".

En realidad, un cuadro de diálogo es una ventana normal, aunque con algunas peculiaridades. También tiene su procedimiento de ventana (procedimiento de diálogo), pero puede devolver un valor a la ventana que lo invoque.

Igual que los menús, los cuadros de diálogo se pueden construir durante la ejecución o a partir de un fichero de recursos.

## Ficheros de recursos

La mayoría de los compiladores de C/C++ que incluyen soporte para Windows poseen herramientas para la edición de recursos: menús, diálogos, bitmaps, etc. Sin embargo considero que es interesante que aprendamos a construir nuestros recursos con un editor de textos, cada compilador tiene sus propios editores de recursos, y no tendría sentido explicar cada uno de ellos. El compilador que usamos "Dev C++", en su versión 4, tiene un editor muy limitado y no aconsejo su uso. De hecho, en la versión actual ya no se incluye, y los ficheros de recursos se editan en modo texto.

De modo que aprenderemos a hacer cuadros de diálogo igual que hemos aprendido a hacer menús: usando el editor de texto.

Para el primer programa de ejemplo de programa con diálogos, que será el ejemplo 4, partiremos de nuevo del programa del ejemplo 1. Nuestro primer diálogo será muy sencillo: un simple cuadro con un texto y un botón de "Aceptar".

Este es el código del fichero de recursos:

```
#include <windows.h>
#include "win004.h"

Menu MENU
BEGIN
    POPUP "&Principal"
    BEGIN
        MENUITEM "&Diálogo", CM_DIALOGO
    END
END

DialogoPrueba DIALOG 0, 0, 118, 48
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
BEGIN
    CONTROL "Mensaje de prueba", TEXTO, "static",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        8, 9, 84, 8
    CONTROL "Aceptar", IDOK, "button",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        56, 26, 50, 14
END
```

Necesitamos incluir el fichero "windows.h" ya que en él se definen muchas constantes, como por ejemplo "IDOK" que es el identificador que se usa para el botón de "Aceptar".

También necesitaremos el fichero "win004.h", para definir los identificadores que usaremos en nuestro programa, por ejemplo el identificador del comando de menú para abrir nuestro diálogo.

```
/* Identificadores */
```

```
/* Identificadores de comandos */  
#define CM_DIALOGO 101  
#define TEXTO      100
```

Lo primero que hemos definido es un menú para poder comunicarle a nuestra aplicación que queremos abrir un cuadro de diálogo.

A continuación está la definición del diálogo, que se compone de varias líneas. Puedes ver más detalles en el apartado de recursos dedicado al [recurso diálogo](#).

De momento bastará con un identificador, como el que usábamos para los menús, y además las coordenadas y dimensiones del diálogo.

En cuanto a los estilos, las constantes para definir los estilos de ventana, que comienzan con "WS\_", puedes verlos con detalle en la sección de constantes "[estilos de ventana](#)". Y los estilos de diálogos, que comienzan con "DS\_", en "[estilos de diálogo](#)".

Para empezar, hemos definido los siguientes estilos:

- **DS\_MODALFRAME**: indica que se creará un cuadro de diálogo con un marco de dialog-box modal que puede combinarse con una barra de título y un menú de sistema.
- **WS\_POPUP**: crea una ventana "pop-up".
- **WS\_VISIBLE**: crea una ventana inicialmente visible.
- **WS\_CAPTION**: crea una ventana con una barra de título, (incluye el estilo **WS\_BORDER**).

La siguiente línea es la de **CAPTION**, en ella especificaremos el texto que aparecerá en la barra de título del diálogo.

La línea de **FONT** sirve para especificar el tamaño y el tipo de fuente de caracteres que usará nuestro diálogo.

Después está la zona de [controles](#), en nuestro ejemplo sólo hemos incluido un texto estático y un botón.

Un control estático (static) nos sirve para mostrar textos o rectángulos, que podemos usar para informar al usuario de algo,

como etiquetas o como adorno. Para más detalles ver [control static](#).

```
CONTROL "Mensaje de prueba", -1, "static",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    8, 9, 84, 8
```

- **CONTROL** es una palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, introducimos el texto que se mostrará.
- id es el identificador del control. Como los controles static no se suelen manejar por las aplicaciones no necesitamos un identificador, así que ponemos -1.
- class es la clase de control, en nuestro caso "static".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un [estilo estático](#) y varios [de ventana](#):
  - **SS\_LEFT**: indica un simple rectángulo y el texto suministrado se alinea en su interior a la izquierda.
  - **WS\_CHILD**: crea el control como una ventana hija.
  - **WS\_VISIBLE**: crea una ventana inicialmente visible.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

El control button nos sirve para comunicarnos con el diálogo, podemos darle comandos del mismo tipo que los que proporciona un menú. Para más detalles ver [recurso button](#).

```
CONTROL "Aceptar", IDOK, "button",
    BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
    WS_TABSTOP,
    56, 26, 50, 14
```

- **CONTROL** es una palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, introducimos el texto que se mostrará en su interior.
- id es el identificador del control. Nuestra aplicación recibirá este identificador junto con el mensaje **WM\_COMMAND** cuando el usuario active el botón. La etiqueta **IDOK** está definida en el fichero Windows.h.
- class es la clase de control, en nuestro caso "button".
- style es el estilo de control que queremos. En nuestro caso es una combinación de varios **estilos de button** y varios **de ventana**:
  - **BS\_PUSHBUTTON**: crea un botón corriente que envía un mensaje **WM\_COMMAND** a su ventana padre cuando el usuario selecciona el botón.
  - **BS\_CENTER**: centra el texto horizontalmente en el área del botón.
  - **WS\_CHILD**: crea el control como una ventana hija.
  - **WS\_VISIBLE**: crea una ventana inicialmente visible.
  - **WS\_TABSTOP**: define un control que puede recibir el foco del teclado cuando el usuario pulsa la tecla TAB. Presionando la tecla TAB, el usuario mueve el foco del teclado al siguiente control con el estilo **WS\_TABSTOP**.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

## Procedimiento de diálogo

Como ya hemos dicho, un diálogo es básicamente una ventana, y al igual que aquella, necesita un procedimiento asociado que procese los mensajes que le sean enviados, en este caso, un procedimiento de diálogo.

## Sintaxis

```
BOOL CALLBACK DialogProc(  
    HWND hwndDlg,    // manipulador del cuadro de diálogo  
    UINT uMsg,       // mensaje  
    WPARAM wParam,   // primer parámetro del mensaje  
    LPARAM lParam    // segundo parámetro del mensaje  
);
```

- hwndDlg identifica el cuadro de diálogo y es el manipulador de la ventana a la que está destinado el mensaje.
- msg es el código del mensaje.
- wParam es el parámetro de tipo palabra asociado al mensaje.
- lParam es el parámetro de tipo doble palabra asociado al mensaje.

La diferencia con el procedimiento de ventana que ya hemos visto está en el tipo de valor de retorno, que es el caso del procedimiento de diálogo es de tipo booleano. Puedes consultar una sintaxis más completa de esta función en [DialogProc](#).

Excepto en la respuesta al mensaje [WM\\_INITDIALOG](#), el procedimiento de diálogo debe retornar con un valor no nulo si procesa el mensaje y cero si no lo hace. Cuando responde a un mensaje [WM\\_INITDIALOG](#), el procedimiento debe retornar cero si llama a la función [SetFocus](#) para poner el foco a uno de los controles del cuadro de diálogo. En otro caso, debe retornar un valor distinto de cero, y el sistema pondrá el foco en el primer control del diálogo que pueda recibirlo.

## Prototipo de procedimiento de diálogo

El prototipo es parecido al de los procedimientos de ventana:

```
BOOL CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM);
```

# Implementación de procedimiento de diálogo para nuestro ejemplo

Nuestro ejemplo es muy sencillo, ya que nuestro diálogo sólo puede proporcionar un comando, así que sólo debemos responder a un tipo de mensaje `WM_COMMAND` y al mensaje `WM_INITDIALOG`.

Según hemos explicado un poco más arriba, del mensaje `WM_INITDIALOG` debemos retornar con un valor distinto de cero si no llamamos a `SetFocus`, como es nuestro caso.

Este mensaje lo usaremos para inicializar nuestro diálogo antes de que sea visible para el usuario, siempre que haya algo que inicializar, claro.

Cuando procesemos el mensaje `WM_COMMAND`, que será siempre el que procede del único botón del diálogo, cerraremos el diálogo llamando a la función `EndDialog` y retornaremos con un valor distinto de cero.

En cualquier otro caso retornamos con `FALSE`, ya que no estaremos procesando el mensaje.

Nuestra función queda así:

```
BOOL CALLBACK DlgProc(HWND hDlg, UINT msg, WPARAM wParam,
LPARAM lParam)
{
    switch (msg)                                /* manipulador del mensaje
*/
    {
        case WM_INITDIALOG:
            return TRUE;
        case WM_COMMAND:
            EndDialog(hDlg, FALSE);
            return TRUE;
    }
    return FALSE;
}
```

Bueno, sólo nos falta saber cómo creamos un cuadro de diálogo. Para ello usaremos un comando de menú, por lo tanto, el diálogo se activará desde el procedimiento de ventana.

```
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT msg, WPARAM
wParam, LPARAM lParam)
{
    static HINSTANCE hInstance;

    switch (msg)                                /* manipulador del mensaje
*/
    {
        case WM_CREATE:
            hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
            return 0;
            break;
        case WM_COMMAND:
            switch(LOWORD(wParam)) {
                case CM_DIALOGO:
                    DialogBox(hInstance, "DialogoPrueba", hwnd,
DlgProc);
                    break;
            }
            break;
        case WM_DESTROY:
            PostQuitMessage(0);    /* envía un mensaje
WM_QUIT a la cola de mensajes */
            break;
        default:                                /* para los mensajes de
los que no nos ocupamos */
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

En este procedimiento hay varias novedades:

Primero hemos declarado una variable estática "hInstance" para tener siempre a mano un manipulador de la instancia actual.

Para inicializar este valor hacemos uso del mensaje [WM\\_CREATE](#), que se envía a una ventana cuando es creada, antes de que se visualice por primera vez. Aprovechamos el hecho de que



nuestro procedimiento de ventana sólo recibe una vez este mensaje y de que lo hace antes de poder recibir ningún otro mensaje o comando. En el futuro veremos que se usa para toda clase de inicializaciones.

El mensaje `WM_CREATE` tiene como parámetro en `lParam` un puntero a una estructura `CREATESTRUCT` que contiene información sobre la ventana. En nuestro caso sólo nos interesa el campo `hInstance`.

La otra novedad es la llamada a la función `DialogBox`, que es la que crea el cuadro de diálogo.

**Nota:**

Bueno, en realidad `DialogBox` no es una función, sino una macro, pero dado su formato y el modo en que se usa, la consideraremos como una función.

Esta *función* necesita varios parámetros:

1. Un manipulador a la instancia de la aplicación, que hemos obtenido al procesar el mensaje `WM_CREATE`.
2. Un identificador de recurso de diálogo, este es el nombre que utilizamos para el diálogo al crear el recurso, entre comillas.
3. Un manipulador a la ventana a la que pertenece el diálogo.
4. Y la dirección del procedimiento de ventana que hará el tratamiento del diálogo.

Y ya tenemos nuestro primer ejemplo del uso de diálogos, en capítulos siguientes empezaremos a conocer más detenidamente cómo usar cada uno de los controles básicos: Edit, List Box, Scroll Bar, Static, Button, Combo Box, Group Box, Check Button y Radio Button. Le dedicaremos un capítulo a cada uno de ellos.

## Pasar parámetros a un cuadro de diálogo

Tenemos otra opción a la hora de crear un diálogo. En lugar de usar la macro [DialogBox](#), podemos usar la función [DialogBoxParam](#), que nos permite enviar un parámetro extra al procedimiento de diálogo. Este parámetro se envía a través del parámetro IParam del procedimiento de diálogo, y puede contener un valor entero, o lo que es mucho más útil, un puntero.

Esta función tiene los mismos parámetros que [DialogBox](#), más uno añadido. Este quinto parámetro es el que podemos usar para pasar y recibir valores desde el procedimiento de diálogo.

Por ejemplo, supongamos que queremos saber cuántas veces se ha invocado un diálogo. Para ello llevaremos la cuenta en el procedimiento de ventana, incrementando esa cuenta cada vez que recibamos un comando para mostrar el diálogo. Además, pasaremos ese valor como parámetro IParam al procedimiento de diálogo.

```
static int veces;
...
case WM_COMMAND:
    switch(LOWORD(wParam)) {
        case CM_DIALOGO:
            DialogBox(hInstance, "DialogoPrueba", hwnd,
DlgProc);
            break;
        case CM_DIALOGO2:
            veces++;
            DialogBoxParam(hInstance, "DialogoPrueba2",
hwnd, DlgProc2, veces);
            break;
    }
    break;
```

Finalmente, nuestro procedimiento de diálogo tomará ese valor y lo usará para crear el texto de un control estático. (Cómo funciona esto lo veremos en otro capítulo, de momento sirva como ejemplo).

```

BOOL CALLBACK DlgProc2(HWND hDlg, UINT msg, WPARAM wParam,
LPARAM lParam)
{
    char texto[25];

    switch (msg)                                /* manipulador del mensaje
*/
    {
        case WM_INITDIALOG:
            sprintf(texto, "Veces invocado: %d", lParam);
            SetWindowText(GetDlgItem(hDlg, TEXT0), texto);
            return TRUE;
        case WM_COMMAND:
            EndDialog(hDlg, FALSE);
            return TRUE;
    }
    return FALSE;
}

```

Hemos usado la función estándar *sprintf* para conseguir un texto estático a partir del parámetro *lParam*. Posteriormente, usamos ese texto para modificar el control estático *TEXT0*.

Usamos la misma plantilla de diálogo para ambos ejemplos, y aprovechamos el control estático para mostrar nuestro mensaje. La función [SetWindowText](#) se usa para cambiar el título de una ventana, pero también sirve para cambiar el texto de un control estático.

Cuando usemos cuadros de diálogo para pedir datos al usuario veremos que este modo de crearlos nos facilita en intercambio de datos entre la aplicación y los procedimientos de diálogo. De otro modo tendríamos que acudir a variables globales.

## Ejemplo 4

# Capítulo 7 Control básico Edit

Tal como hemos definido nuestro diálogo en el capítulo 6, no tiene mucha utilidad. Los diálogos se usan para intercambiar información entre la aplicación y el usuario, en ambas direcciones. El ejemplo 4 sólo lo hace en una de ellas.

En el capítulo anterior hemos usado dos controles (un texto estático y un botón), aunque sin saber exactamente cómo funcionan. En este capítulo veremos el uso del control de edición.

Un control edit es una ventana de control rectangular que permite al usuario introducir y editar texto desde el teclado.

Cuando está seleccionado muestra el texto que contiene y un cursor intermitente que indica el punto de inserción de texto. Para seleccionarlo el usuario puede hacer un click con el ratón en su interior o usar la tecla [TAB]. El usuario podrá entonces introducir texto, cambiar el punto de inserción, o seleccionar texto para ser borrado o movido usando el teclado o el ratón.

Un control de este tipo puede enviar mensajes a su ventana padre mediante [WM\\_COMMAND](#), y la ventana padre puede enviar mensajes a un control edit en un cuadro de diálogo llamando a la función [SendDlgItemMessage](#). Veremos algunos de estos mensajes en este capítulo, y el resto en los capítulos más avanzados.

## Fichero de recursos

Empezaremos definiendo el control edit en el fichero de recursos, y lo añadiremos a nuestro dialogo de prueba.

```
#include <windows.h>
#include "win005.h"
```

```

Menu MENU
BEGIN
    POPUP "&Principal"
    BEGIN
        MENUITEM "&Diálogo", CM_DIALOGO
    END
END

DialogoPrueba DIALOG 0, 0, 118, 48
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
BEGIN
    CONTROL "Texto:", -1, "STATIC",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        8, 9, 28, 8
    CONTROL "", ID_TEXTO, "EDIT",
        ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_TABSTOP,
        36, 9, 76, 12
    CONTROL "Aceptar", IDOK, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        56, 26, 50, 14
END

```

Hemos hecho algunas modificaciones más. Para empezar, el control static se ha convertido en una etiqueta para el control edit, que indica al usuario qué tipo de información debe suministrar.

Hemos añadido el control edit a continuación del control static. Veremos que el orden en que aparecen los controles dentro del cuadro de diálogo es muy importante, al menos en aquellos controles que tengan el estilo **WS\_TABSTOP**, ya que ese orden será el mismo en que se activen los controles cuando usemos la tecla TAB. Para más detalles acerca de los controles edit ver [controles edit](#).

Pero ahora veamos cómo hemos definido nuestro control edit:

```

CONTROL "", ID_TEXTO, "EDIT",
    ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
    36, 9, 76, 12

```

- **CONTROL** es una palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, introducimos el texto que se mostrará en el interior del control, en este caso, ninguno.
- id es el identificador del control. Los controles edit necesitan un identificador para que la aplicación pueda acceder a ellos. Usaremos un identificador definido en win005.h.
- class es la clase de control, en nuestro caso "EDIT".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un **estilo edit** y varios **de ventana**:
  - **ES\_LEFT**: indica que el texto en el interior del control se alineará a la izquierda.
  - **WS\_CHILD**: crea el control como una ventana hija.
  - **WS\_VISIBLE**: crea una ventana inicialmente visible.
  - **WS\_BORDER**: se crea un control que tiene de borde una línea fina.
  - **WS\_TABSTOP**: define un control que puede recibir el foco del teclado cuando el usuario pulsa la tecla TAB. Presionando la tecla TAB, el usuario mueve el foco del teclado al siguiente control con el estilo **WS\_TABSTOP**.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

## El procedimiento de diálogo y los controles edit

Para manejar el control edit desde nuestro procedimiento de diálogo tendremos que hacer algunas modificaciones.

Para empezar, los controles edit también pueden generar mensajes **WM\_COMMAND**, de modo que debemos diferenciar el control que originó dicho mensaje y tratarlo de diferente modo según el caso.

```
case WM_COMMAND:
    if(LOWORD(wParam) == IDOK) EndDialog(hDlg,
FALSE);
    return TRUE;
```

En nuestro caso sigue siendo sencillo: sólo cerraremos el diálogo si el mensaje `WM_COMMAND` proviene del botón "Aceptar".

La otra modificación afecta al mensaje `WM_INITDIALOG`.

```
case WM_INITDIALOG:
    SetFocus(GetDlgItem(hDlg, ID_TEXTO));
    return FALSE;
```

De nuevo es una modificación sencilla, tan sólo haremos que el foco del teclado se coloque en el control edit, de modo que el usuario pueda empezar a escribir directamente, tan pronto como el diálogo haya aparecido en pantalla.

Para hacer eso usaremos la función `SetFocus`. Pero esta función requiere como parámetro el manipulador de ventana del control que debe recibir el foco, este manipulador lo conseguimos con la función `GetDlgItem`, que a su vez necesita como parámetros un manipulador del diálogo y el identificador del control.

## Variables a editar en los cuadros de diálogo

Quizás has notado que a nuestro programa le falta algo.

Efectivamente, podemos introducir y modificar texto en el cuadro de diálogo, pero no podemos asignar valores iniciales al control de edición ni tampoco podemos hacer que la aplicación tenga acceso al texto introducido por el usuario.

Lo primero que tenemos que tener es algún tipo de variable que puedan compartir los procedimientos de ventana de la aplicación y el del diálogo. En nuestro caso se trata sólo de una cadena, pero

según se añadan más parámetros al cuadro de edición, estos datos pueden ser más complejos, así que usaremos un sistema que nos valdrá en todos los casos.

Se trata de crear una estructura con todos los datos que queremos que el procedimiento de diálogo comparta con el procedimiento de ventana:

```
typedef struct stDatos {  
    char Texto[80];  
} DATOS;
```

Lo más sencillo es que estos datos sean globales, pero no será buena idea ya que no es buena práctica el uso de variables globales.

Tampoco parece muy buena idea declarar los datos en el procedimiento de ventana, ya que este procedimiento se usa para todas las ventanas de la misma clase, y tendríamos que definir los datos como estáticos.

Pero recordemos que tenemos un modo de pasar parámetros al cuadro de diálogo, usando la función [DialogBoxParam](#), a través del parámetro *lParam*.

Aunque esta opción parece que nos limita a valores enteros, y sólo permite pasar valores al procedimiento de diálogo, en realidad se puede usar para pasar valores en ambos sentidos, bastará con enviar un puntero en lugar de un entero.

Para ello haremos un casting del puntero al tipo [LPARAM](#). Dentro del procedimiento de diálogo haremos otro casting de [LPARAM](#) al puntero.

Esto nos permite declarar la variable que contiene los datos dentro del procedimiento de ventana, en este caso, de forma estática.

```
static DATOS Datos;
```



```
...
    DialogBoxParam(hInstance, "DialogoPrueba", hwnd, DlgProc,
(LPARAM) &Datos);
```

En el caso del procedimiento de diálogo:

```
static DATOS *Datos;
...
    case WM_INITDIALOG:
        Datos = (DATOS *)lParam;
```

Daremos valores iniciales a las variables de la aplicación, dentro del procedimiento de ventana, al procesar el mensaje [WM\\_CREATE](#):

```
case WM_CREATE:
    /* Inicialización de los datos de la aplicación
*/
    strcpy(Datos.Texto, "Inicial");
```

## Iniciar controles edit

Ahora tenemos que hacer que se actualice el contenido del control edit al abrir el cuadro de diálogo.

El lugar adecuado para hacer esto es en el proceso del mensaje [WM\\_INITDIALOG](#):

```
case WM_INITDIALOG:
    SendDlgItemMessage(hDlg, ID_TEXTO, EM_LIMITTEXT,
80, 0L);
    Datos = (DATOS *)lParam;
    SetDlgItemText(hDlg, ID_TEXTO, Datos->Texto);
    SetFocus(GetDlgItem(hDlg, ID_TEXTO));
    return FALSE;
```

Hemos añadido dos llamadas a dos nuevas funciones del API. La primera es a [SendDlgItemMessage](#), que envía un mensaje a un control. En este caso se trata de un mensaje [EM\\_LIMITTEXT](#), que sirve para limitar la longitud del texto que se puede almacenar y editar en el control edit. Es necesario que hagamos esto, ya que el texto que puede almacenar nuestra estructura de datos está limitado a 80 caracteres.

También hemos añadido una llamada a la función [SetDlgItemText](#), que hace exactamente lo que pretendemos: cambiar el contenido del texto en el interior de un control edit.

## Devolver valores a la aplicación

También queremos que cuando el usuario esté satisfecho con los datos que ha introducido, y pulse el botón de aceptar, el dato de nuestra aplicación se actualice con el texto que hay en el control edit.

Esto lo podemos hacer de varios modos. Como veremos en capítulos más avanzados, podemos responder a mensajes que provengan del control cada vez que cambia su contenido.

Pero ahora nos limitaremos a leer ese contenido cuando procesemos el comando generado al pulsar el botón de "Aceptar".

```
case WM_COMMAND:
    if (LOWORD(wParam) == IDOK)
    {
        GetDlgItemText(hDlg, ID_TEXTO, Datos->Texto,
80);
        EndDialog(hDlg, FALSE);
    }
    return TRUE;
```

Para eso hemos añadido la llamada a la función [GetDlgItemText](#), que es simétrica a [SetDlgItemText](#).

Ahora puedes comprobar lo que pasa cuando abres varias veces seguidas el cuadro de diálogo modificando el texto cada vez.

Con esto parece que ya controlamos lo básico de los controles edit, pero aún hay algo más.

## Añadir la opción de cancelar

Es costumbre dar al usuario la oportunidad de arrepentirse si ha modificado algo en un cuadro de diálogo y, por la razón que sea, cambia de idea.

Para eso se suele añadir un segundo botón de "Cancelar".

Empecemos por añadir dicho botón en el fichero de recursos:

```
DialogoPrueba DIALOG 0, 0, 118, 48
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
{
    CONTROL "Texto:", -1, "static",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        8, 9, 28, 8
    CONTROL "", ID_TEXTO, "EDIT",
        ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_TABSTOP,
        36, 9, 76, 12
    CONTROL "Aceptar", IDOK, "BUTTON",
        BS_DEFPUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        8, 26, 45, 14
    CONTROL "Cancelar", IDCANCEL, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        61, 26, 45, 14
}
```

Hemos cambiado las coordenadas de los botones, para que el de "Aceptar" aparezca a la izquierda. Además, el botón de "Aceptar" lo hemos convertido en el botón por defecto, añadiendo el estilo

**BS\_DEFPUSHBUTTON**. Haciendo eso, podemos simular la pulsación del botón de aceptar pulsando la tecla de "intro".

El identificador del botón de "Cancelar" es **IDCANCEL**, y está definido en Windows.h.

Ahora tenemos que hacer que nuestro procedimiento de diálogo manipule el mensaje del botón de "Cancelar".

```
case WM_COMMAND:
    switch (LOWORD(wParam)) {
        case IDOK:
            GetDlgItemText(hDlg, ID_TEXTO, Datos-
>Texto, 80);
            EndDialog(hDlg, FALSE);
            break;
        case IDCANCEL:
            EndDialog(hDlg, FALSE);
            break;
    }
    return TRUE;
```

Como puedes ver, sólo leemos el contenido del control edit si se ha pulsado el botón de "Aceptar".

## Ejemplo 5

### Editar números

En muchas ocasiones necesitaremos editar valores de números enteros en nuestros diálogos.

Para eso, el API tiene previstas algunas constantes y funciones, (aunque no es así para números en coma flotante, para los que tendremos que crear nuestros propios controles).

Bien, vamos a modificar nuestro ejemplo para editar valores numéricos en lugar de cadenas de texto.

## Fichero de recursos para editar enteros

Empezaremos añadiendo una constante al fichero de identificadores: "win006.h":

```
#define ID_NUMERO 100
```

Y redefiniendo el control edit en el fichero de recursos, al que añadiremos el flag **ES\_NUMBER** para que sólo admita caracteres numéricos:

```
DialogoPrueba DIALOG 0, 0, 118, 48
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
BEGIN
    CONTROL "Número:", -1, "STATIC",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        8, 9, 28, 8
    CONTROL "", ID_NUMERO, "EDIT",
        ES_NUMBER | ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER
    | WS_TABSTOP,
        36, 9, 76, 12
    CONTROL "Aceptar", IDOK, "BUTTON",
        BS_DEFPUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        8, 26, 45, 14
    CONTROL "Cancelar", IDCANCEL, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        61, 26, 45, 14
END
```

## Variables a editar en los cuadros de diálogo

Ahora modificaremos la estructura de los datos para que el dato a editar sea de tipo numérico:

```
typedef struct stDatos {
    int Numero;
} DATOS;
```

Al igual que antes, daremos valores iniciales a las variables del diálogo al procesar el mensaje [WM\\_CREATE](#).

```
        case WM_CREATE:
            /* Inicialización de los datos de la aplicación
*/
            Datos.Numero = 123;
```

Por supuesto, pasaremos un puntero a esta estructura a la función [DialogBoxParam](#), haciendo uso el parámetro *lParam*:

```
static DATOS Datos;
...
DialogBoxParam(hInstance, "DialogoPrueba", hwnd, DlgProc,
(LPARAM)&Datos);
```

## Iniciar controles edit de enteros

Ahora tenemos que hacer que se actualice el contenido del control edit al abrir el cuadro de diálogo.

El lugar adecuado para hacer esto es en el proceso del mensaje [WM\\_INITDIALOG](#):

```
static DATOS *datos;
...
        case WM_INITDIALOG:
            datos = (DATOS *)lParam;
            SetDlgItemInt(hDlg, ID_NUMERO, (UINT)datos->Numero, FALSE);
```

```
SetFocus(GetDlgItem(hDlg, ID_NUMERO));  
return FALSE;
```

En este caso no es necesario limitar el texto que podemos editar en el control, ya que, como veremos, las propias funciones del API se encargan de capturar y convertir el contenido del control en un número, de modo que no tenemos que preocuparnos de que no quepa en nuestra variable.

También hemos modificado la función a la que llamamos para modificar el contenido del control, ahora usaremos [SetDlgItemInt](#), que cambia el contenido de un control edit con un valor numérico.

## Devolver valores a la aplicación

Por último leeremos el contenido cuando procesemos el comando generado al pulsar el botón de "Aceptar".

```
BOOL NumeroOk;  
int numero;  
...  
case WM_COMMAND:  
    switch(LOWORD(wParam)) {  
        case IDOK:  
            numero = GetDlgItemInt(hDlg, ID_NUMERO,  
&NumeroOk, FALSE);  
            if(NumeroOk) {  
                datos->Numero = numero;  
                EndDialog(hDlg, FALSE);  
            }  
            else  
                MessageBox(hDlg, "Número no válido",  
"Error",  
                MB_ICONEXCLAMATION | MB_OK);  
            break;
```

Para eso hemos añadido la llamada a la función [GetDlgItemInt](#), que es simétrica a [SetDlgItemInt](#). El proceso difiere del usado para

capturar cadenas, ya que en este caso la función nos devuelve el valor numérico del contenido del control edit.

También devuelve un parámetro que indica si ha habido algún error durante la conversión. Si el valor de ese parámetro es **TRUE**, significa que la conversión se realizó sin problemas, si es **FALSE**, es que ha habido un error. Si nuestro programa detecta un error visualizará un mensaje de error y no permitirá abandonar el cuadro de diálogo.

Pero si ha habido un error, el valor de retorno de **GetDlgItemInt** será cero. Esto nos causa un problema. Si leemos el valor directamente en `datos->Numero` y el usuario introduce un valor no válido, y después pulsa "Cancelar", el valor devuelto no será el original, sino 0. Para evitar eso hemos usado una variable local, y el valor de `datos->Numero` sólo se actualiza antes de salir con "Aceptar" y con un valor válido.

Por último, hemos usado el flag **BM\_ICONEXCLAMATION** en el **MessageBox**, que añade un icono al cuadro de mensaje y el sonido predeterminado para alertar al usuario.

## Ejemplo 6



# Capítulo 8 Control básico

## ListBox

Los controles edit son muy útiles cuando la información a introducir por el usuario es imprevisible o existen muchas opciones. Pero cuando el número de opciones no es muy grande y son todas conocidas, es preferible usar un control ListBox.

Ese es el siguiente control básico que veremos. Un ListBox consiste en una ventana rectangular con una lista de cadenas entre las cuales el usuario puede escoger una o varias.

El usuario puede seleccionar una cadena apuntándola y haciendo clic con el botón del ratón. Cuando una cadena se selecciona, se resalta y se envía un mensaje de notificación a la ventana padre. También se puede usar una barra de scroll con los listbox para desplazar listas muy largas o demasiado anchas para la ventana.

## Ficheros de recursos

Empezaremos definiendo el control listbox en el fichero de recursos, y lo añadiremos a nuestro diálogo de prueba:

```
#include <windows.h>;
#include "win007.h"

Menu MENU
BEGIN
    POPUP "&Principal"
    BEGIN
        MENUITEM "&Diálogo", CM_DIALOGO
    END
END
```

```

DialogoPrueba DIALOG 0, 0, 118, 135
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
BEGIN
    CONTROL "Lista:", -1, "static",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        8, 9, 28, 8
    CONTROL "", ID_LISTA, "listbox",
        LBS_STANDARD | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        9, 19, 104, 99
    CONTROL "Aceptar", IDOK, "BUTTON",
        BS_DEFPUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        8, 116, 45, 14
    CONTROL "Cancelar", IDCANCEL, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        61, 116, 45, 14
END

```

Hemos añadido el control listbox a continuación del control static. Para más detalles acerca de los controles listbox ver [control listbox](#). Ahora veamos cómo hemos definido nuestro control listbox:

```

CONTROL "", ID_LISTA, "listbox",
    LBS_STANDARD | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
    9, 19, 104, 99

```

- **CONTROL** es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los listbox no tiene ninguna función. Lo dejaremos como cadena vacía.
- id es el identificador del control. Los controles listbox necesitan un identificador para que la aplicación pueda acceder a ellos. Usaremos un identificador definido en win007.h.
- class es la clase de control, en nuestro caso "LISTBOX".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un [estilo listbox](#) y varios [de ventana](#):

- **LBS\_STANDARD**: ordena alfabéticamente las cadenas en el listbox. La ventana padre recibe un mensaje de entrada cada vez que el usuario hace click o doble click sobre una cadena. El list box tiene bordes en todos sus lados.
  - **WS\_CHILD**: crea el control como una ventana hija.
  - **WS\_VISIBLE**: crea una ventana inicialmente visible.
  - **WS\_TABSTOP**: define un control que puede recibir el foco del teclado cuando el usuario pulsa la tecla TAB. Presionando la tecla TAB, el usuario mueve el foco del teclado al siguiente control con el estilo **WS\_TABSTOP**.
- coordenada x del control.
  - coordenada y del control.
  - width: anchura del control.
  - height: altura del control.

## Iniciar controles listbox

Para este ejemplo también usaremos variables estáticas en el procedimiento de ventana para almacenar el valor de la cadena del listbox actualmente seleccionada.

```
// Datos de la aplicación
typedef struct stDatos {
    char Item[80];
} DATOS;
```

Daremos valores iniciales a las variables, al procesar el mensaje **WM\_CREATE** del procedimiento de ventana:

```
static DATOS Datos;
...
case WM_CREATE:
    hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
    /* Inicialización de los datos de la aplicación
*/
```

```
strcpy(Datos.Item, "Cadena nº 3");  
return 0;
```

Y pasaremos un puntero a la estructura con los datos como parámetro *lParam* de la función [DialogBoxParam](#).

```
DialogBoxParam(hInstance, "DialogoPrueba",  
hwnd,  
DlgProc, (LPARAM) &Datos);
```

La característica más importante de los listbox es que contienen listas de cadenas. Así que es imprescindible iniciar este tipo de controles, introduciendo las cadenas antes de que se muestre el diálogo. Eso se hace durante el proceso del mensaje [WM\\_INITDIALOG](#) dentro del procedimiento de diálogo. En este mismo mensaje obtenemos el puntero a la estructura de los datos que recibimos en el parámetro *lParam*.

```
static DATOS *Datos;  
...  
case WM_INITDIALOG:  
    Datos = (DATOS *)lParam;  
    // Añadir cadenas. Mensaje: LB_ADDSTRING  
    SendDlgItemMessage(hDlg, ID_LISTA, LB_ADDSTRING,  
0, (LPARAM) "Cadena nº 1");  
    SendDlgItemMessage(hDlg, ID_LISTA, LB_ADDSTRING,  
0, (LPARAM) "Cadena nº 4");  
    SendDlgItemMessage(hDlg, ID_LISTA, LB_ADDSTRING,  
0, (LPARAM) "Cadena nº 3");  
    SendDlgItemMessage(hDlg, ID_LISTA, LB_ADDSTRING,  
0, (LPARAM) "Cadena nº 2");  
    SendDlgItemMessage(hDlg, ID_LISTA,  
LB_SELECTSTRING, (UINT)-1, (LPARAM) Datos->Item);  
    SetFocus(GetDlgItem(hDlg, ID_LISTA));  
    return FALSE;
```

Para añadir cadenas a un listbox se usa el mensaje **LB\_ADDSTRING** mediante la función **SendDlgItemMessage**, que envía un mensaje a un control.

También podemos preseleccionar alguna de las cadenas del listbox, aunque esto no es muy frecuente ya que se suele dejar al usuario que seleccione una opción sin sugerirle nada. Para seleccionar una de las cadenas también se usa un mensaje: **LB\_SELECTSTRING**. Usaremos el valor -1 en wParam para indicar que se busque en todo el listbox.

## Devolver valores a la aplicación

También queremos que cuando el usuario está satisfecho con los datos que ha introducido, y pulse el botón de aceptar, el dato de nuestra aplicación se actualice con el texto del ítem seleccionado.

De nuevo recurriremos a mensajes para pedirle al listbox el valor de la cadena actualmente seleccionada. En este caso se trata dos mensajes combinados, uno es **LB\_GETCURSEL**, que se usa para averiguar el índice de la cadena actualmente seleccionada. El otro es **LB\_GETTEXT**, que devuelve la cadena del índice que le indiquemos.

Cuando trabajemos con memoria dinámica y con ítems de longitud variable, será interesante saber la longitud de la cadena antes de leerla desde el listbox. Para eso podemos usar el mensaje **LB\_GETTEXTLEN**.

Haremos esa lectura al procesar el comando **IDOK**, que se genera al pulsar el botón "Aceptar".

```
UINT indice;
...
case WM_COMMAND:
    switch(LOWORD(wParam)) {
        case IDOK:
            indice = SendDlgItemMessage(hDlg, ID_LISTA,
LB_GETCURSEL, 0, 0);
```

```
        SendDlgItemMessage(hDlg, ID_LISTA,
LB_GETTEXT, indice, (LPARAM)Datos->Item);
        EndDialog(hDlg, FALSE);
        break;
    case IDCANCEL:
        EndDialog(hDlg, FALSE);
        break;
    }
    return TRUE;
```

## Ejemplo 7

# Capítulo 9 Control básico

## Button

Los controles button simulan el comportamiento de un pulsador o un interruptor. Pero sólo cuando se comportan como un pulsador los llamaremos botones, cuando emulen interruptores nos referiremos a ellos como checkbox o radiobutton.

Los botones se usan para que el usuario pueda ejecutar ciertas acciones o para dar órdenes a una aplicación. En muchos aspectos, funcionan igual que los menús, y de hecho, ambos generan mensajes de tipo [WM\\_COMMAND](#).

Se componen normalmente de una pequeña área rectangular con un texto en su interior que identifica la acción que tienen asociada.

En realidad ya hemos usado controles button en todos los ejemplos anteriores, pero los explicaremos ahora con algo más de detalle.

## Ficheros de recursos

Empezaremos definiendo el control button en el fichero de recursos, y lo añadiremos a nuestro dialogo de prueba:

```
#include <windows.h>
#include "win008.h"

Menu MENU
BEGIN
    POPUP "&Principal"
    BEGIN
        MENUITEM "&Diálogo", CM_DIALOGO
```

```

END
END

DialogoPrueba DIALOG 0, 0, 130, 70
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
BEGIN
    CONTROL "Botones:", -1, "static",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        8, 9, 28, 8
    CONTROL "Nuestro botón", ID_BOTON, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        9, 19, 104, 25
    CONTROL "Aceptar", IDOK, "BUTTON",
        BS_DEFPUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        8, 50, 45, 14
    CONTROL "Cancelar", IDCANCEL, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        61, 50, 45, 14
END

```

Hemos añadido un nuevo control button a continuación del control static. Para más detalles acerca de los controles button ver [controles button](#).

Ahora veamos cómo hemos definido nuestro control button, y también los otros dos que hemos usado hasta ahora.:

```

CONTROL "Nuestro botón", ID_BOTON, "BUTTON",
    BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
    9, 19, 104, 25
CONTROL "Aceptar", IDOK, "BUTTON",
    BS_DEFPUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
    8, 50, 45, 14

```



- **CONTROL** es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los button se trata del texto que aparecerá en su interior.
- id es el identificador del control. Los controles button necesitan un identificador para que la aplicación pueda acceder a ellos y para usarlos como parámetro en los mensajes **WM\_COMMAND**. Usaremos un identificador definido en win008.h.
- class es la clase de control, en nuestro caso "BUTTON".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un **estilo button** y varios **de ventana**:
  - **BS\_PUSHBUTTON**: Crea un botón corriente que envía un mensaje **WM\_COMMAND** a su ventana padre cuando el usuario pulsa el botón.
  - **BS\_DEFPUSHBUTTON**: Crea un botón normal que se comporta como uno del estilo **BS\_PUSHBUTTON**, pero también tiene un borde negro y grueso. Si el botón está en un cuadro de diálogo, el usuario puede pulsar este botón usando la tecla ENTER, aún cuando el botón no tenga el foco de entrada. Este estilo es corriente para permitir al usuario seleccionar rápidamente la opción más frecuente, la opción por defecto. Lo usaremos frecuentemente con el botón "Aceptar".
  - **BS\_CENTER**: Centra el texto horizontalmente en el área del botón.
  - **WS\_CHILD**: crea el control como una ventana hija.
  - **WS\_VISIBLE**: crea una ventana inicialmente visible.
  - **WS\_TABSTOP**: define un control que puede recibir el foco del teclado cuando el usuario pulsa la tecla TAB. Presionando la tecla TAB, el usuario mueve el foco del teclado al siguiente control con el estilo **WS\_TABSTOP**.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

## Iniciar controles button

Los controles button no se usan para editar o seleccionar información, sólo para que el usuario pueda dar órdenes o indicaciones a la aplicación, así que no requieren inicialización. Lo más que haremos en algunos casos es situar el foco en uno de ellos.

Eso se hace durante el proceso del mensaje `WM_INITDIALOG` dentro del procedimiento de diálogo.

```
case WM_INITDIALOG:
    SetFocus(GetDlgItem(hDlg, ID_BOTON));
    return FALSE;
```

## Tratamiento de acciones de los controles button

Nuestro cuadro de diálogo tiene tres botones. Los de "Aceptar" y "Cancelar" tienen una misión clara: validar o ignorar los datos y cerrar el cuadro de diálogo. En el caso de nuestro botón, queremos que se realice otra operación diferente, por ejemplo, mostrar un mensaje.

```
case WM_COMMAND:
    switch(LOWORD(wParam)) {
        case ID_BOTON:
            MessageBox(hDlg, "Se pulsó 'Nuestro botón'", "Acción", MB_ICONINFORMATION|MB_OK);
            break;
        case IDOK:
            EndDialog(hDlg, FALSE);
            break;
        case IDCANCEL:
            EndDialog(hDlg, FALSE);
            break;
```

```
}  
return TRUE;
```

## Ejemplo 8

# Capítulo 10 Control básico Static

Los static son el tipo de control menos interactivo, normalmente se usan como información o decoración. Pero son muy importantes. Windows es un entorno gráfico, y la apariencia forma una parte muy importante de él.

Existen varios tipos de controles static, o mejor dicho, varios estilos de controles static.

Dependiendo del estilo que elijamos para cada control static, su aspecto será radicalmente distinto, desde una simple línea o cuadro hasta un bitmap, un icono o un texto.

Cuando hablemos de controles static de tipo texto, frecuentemente nos referiremos a ellos como etiquetas. Las etiquetas pueden tener también una función añadida, como veremos más adelante: nos servirán para acceder a otros controles usando el teclado.

En realidad ya hemos usado controles static del tipo etiqueta cuando vimos los controles edit, listbox y button, pero de nuevo los explicaremos ahora con más detalle.

## Ficheros de recursos

Empezaremos definiendo varios controles static en el fichero de recursos, y los añadiremos a nuestro dialogo de prueba, para obtener un muestrario:

```
#include <windows.h>
#include "win009.h"
```

```

Menu MENU
BEGIN
    POPUP "&Principal"
    BEGIN
        MENUITEM "&Diálogo", CM_DIALOGO
    END
END

DialogoPrueba DIALOG 0, 0, 240, 120
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Prueba de static"
FONT 8, "Helv"
BEGIN
    CONTROL "Frame1", -1, "STATIC",
        SS_WHITEFRAME | WS_CHILD | WS_VISIBLE,
        8, 5, 52, 34
    CONTROL "Frame2", -1, "STATIC",
        SS_GRAYFRAME | WS_CHILD | WS_VISIBLE,
        12, 9, 52, 34
    CONTROL "Frame3", -1, "STATIC",
        SS_BLACKFRAME | WS_CHILD | WS_VISIBLE,
        16, 13, 52, 34
    CONTROL "Rect1", -1, "STATIC",
        SS_BLACKRECT | WS_CHILD | WS_VISIBLE,
        72, 22, 48, 34
    CONTROL "Rect2", -1, "STATIC",
        SS_GRAYRECT | WS_CHILD | WS_VISIBLE,
        12, 60, 52, 34
    CONTROL "Rect3", -1, "STATIC",
        SS_WHITERECT | WS_CHILD | WS_VISIBLE,
        72, 60, 48, 34
    CONTROL "Bitmap1", -1, "STATIC",
        SS_BITMAP | WS_CHILD | WS_VISIBLE,
        128, 22, 18, 15
    CONTROL "Icono", -1, "STATIC",
        SS_ICON | WS_CHILD | WS_VISIBLE,
        188, 47, 20, 20
    CONTROL "Edit &1:", -1, "STATIC",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        128, 73, 40, 9
    CONTROL "", ID_EDIT1, "EDIT",
        ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_TABSTOP,
        180, 73, 20, 12
    CONTROL "Edit &2:", -1, "STATIC",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        128, 95, 28, 8
    CONTROL "", ID_EDIT2, "EDIT",

```

```

        ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_TABSTOP,
        180, 95, 20, 12
        CONTROL "Aceptar", IDOK, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        186, 6, 50, 14
END

```

Hemos añadido diez nuevos controles static. Para más detalles acerca de los controles static ver **CONTROL#CONTROL\_STATIC:controles static**.

Ahora veamos cómo hemos definido nuestros controles static:

```

CONTROL "Frame1", -1, "static", SS_WHITEFRAME | WS_CHILD |
WS_VISIBLE, 8, 5, 52, 34
CONTROL "Bitmap1", -1, "static", SS_BITMAP | WS_CHILD |
WS_VISIBLE, 128, 22, 18, 15
CONTROL "Icono", -1, "static", SS_ICON | WS_CHILD |
WS_VISIBLE, 188, 47, 20, 20
CONTROL "Edit &1:", -1, "static", SS_LEFT | WS_CHILD |
WS_VISIBLE, 128, 73, 40, 9

```

- **CONTROL** es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los static tiene sentido para las etiquetas, los bitmaps y los iconos. En estos dos últimos casos indicará el nombre del recurso a insertar. En el resto de los casos se incluye como información. Comentaremos algo más sobre los textos de la etiquetas más abajo.
- id es el identificador del control. Los controles static no suelen necesitar un identificador, ya que no suelen tener un comportamiento interactivo. De modo que todos los identificadores de controles static serán -1.
- class es la clase de control, en nuestro caso "STATIC".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un [estilo static](#) y varios [de ventana](#).

- SS\_WHITEFRAME, SS\_GRAYFRAME, SS\_BLACKFRAME: Crea un rectángulo vacío o un marco de color blanco, gris o negro, respectivamente.
- SS\_WHITEREC, SS\_GRAYREC, SS\_BLACKREC: Crea un rectángulo relleno de color blanco, gris o negro, respectivamente.
- SS\_BITMAP mostrará el mapa de bits indicado en el campo text.
- SS\_ICON mostrará el icono indicado en el campo text.
- SS\_LEFT, SS\_RIGHT, SS\_CENTER: indican que es una etiqueta y ajustará el texto a la izquierda, la derecha o el centro, respectivamente.
- WS\_CHILD: crea el control como una ventana hija.
- WS\_VISIBLE: crea una ventana inicialmente visible.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

En el caso de las etiquetas, cuando se incluye el carácter '&', el siguiente carácter de la cadena aparecerá subrayado, indicando que puede ser usado como acelerador, (pulsando la tecla [ALT] más el carácter subrayado), para acceder al control más cercano, normalmente a su derecha o debajo. Pero, cuidado, en realidad el acelerador situará el foco en el control definido exactamente a continuación del control static en el fichero de recursos, y no al más cercano físicamente en pantalla.

En el ejemplo, verifica lo que sucede al pulsar la tecla ALT más '1' ó '2'. Verás que el foco del teclado se desplaza a los cuadros de edición 1 y 2.

## Iniciar controles static

Los controles static normalmente no necesitan iniciarse, por algo son estáticos. Sin embargo, a veces necesitaremos modificar el texto de alguna etiqueta, esto puede ser útil para mostrar alguna

información en un cuadro de diálogo, por ejemplo. Para eso podemos usar la misma función que en los controles edit: [SetDlgItemText](#). En este caso, necesitaremos usar un identificador válido para el control estático.

## **Tratamiento de acciones de los controles static**

Los controles static tampoco responderán, normalmente, a acciones del usuario, ni tampoco generarán mensajes. En el caso de las etiquetas, el comportamiento de los aceleradores es automático y no requerirá ninguna acción del programa.

### **Ejemplo 9**



# Capítulo 11 Control básico

## ComboBox

Los ComboBoxes son una combinación de un control Edit y un ListBox. Son los controles que suelen recordar las entradas que hemos introducido antes, para que podamos seleccionarlas sin tener que escribirlas de nuevo, en ese sentido funcionan igual que un ListBox, pero también permiten introducir nuevas entradas.

Hay modalidades de ComboBox en las que el control Edit está inhibido, y no permite introducir nuevos valores. En esos casos, el control se comportará de un modo muy parecido al que lo hace un ListBox, pero, como veremos más adelante, tienen ciertas ventajas.

Existen tres tipos de ComboBoxes:

- Simple: es la forma que muestra siempre el control Edit y el ListBox, aunque ésta esté vacía.
- DropDown: despliegue hacia abajo. Se muestra un pequeño icono a la derecha del control Edit. Si el usuario lo pulsa con el ratón, se desplegará el ListBox, mientras no se pulse, la lista permanecerá oculta.
- DropDownList: lo mismo, pero el control Edit se sustituye por un control Static.

## Ficheros de recursos

Para nuestro ejemplo incluiremos un control ComboBox de cada tipo, así veremos las peculiaridades de cada uno:

```
#include <windows.h>
#include "win010.h"
```

```

Menu MENU
BEGIN
    POPUP "&Principal"
    BEGIN
        MENUITEM "&Diálogo", CM_DIALOGO
    END
END

DialogoPrueba DIALOG 0, 0, 205, 78
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
CAPTION "Combo boxes"
FONT 8, "Helv"
BEGIN
    CONTROL "&Simple", -1, "static",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        8, 2, 60, 8
    CONTROL "ComboBox1", ID_COMBOBOX1, "COMBOBOX",
        CBS_SORT | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        8, 13, 60, 43
    CONTROL "&Dropdown", -1, "static",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        73, 2, 60, 8
    CONTROL "ComboBox2", ID_COMBOBOX2, "COMBOBOX",
        CBS_DROPDOWN | CBS_SORT | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        72, 13, 60, 103
    CONTROL "Dropdown &List", -1, "static",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        138, 2, 60, 8
    CONTROL "ComboBox3", ID_COMBOBOX3, "COMBOBOX",
        CBS_DROPDOWNLIST | CBS_SORT | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        136, 13, 60, 103
    CONTROL "Aceptar", IDOK, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        28, 60, 50, 14
    CONTROL "Cancelar", IDCANCEL, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        116, 60, 50, 14
END

```

Hemos añadido los nuevos controles ComboBox. Para más detalles acerca de estos controles ver [controles combobox](#).

Ahora veremos más detalles sobre los estilos de los controles  
ComboBox:

```
CONTROL "ComboBox1", ID_COMBOBOX1, "COMBOBOX",  
    CBS_SORT | WS_CHILD | WS_VISIBLE | WS_TABSTOP,  
    8, 13, 60, 43  
CONTROL "ComboBox2", ID_COMBOBOX2, "COMBOBOX",  
    CBS_DROPDOWN | CBS_SORT | WS_CHILD | WS_VISIBLE |  
WS_TABSTOP,  
    72, 13, 60, 103  
CONTROL "ComboBox3", ID_COMBOBOX3, "COMBOBOX",  
    CBS_DROPDOWNLIST | CBS_SORT | WS_CHILD | WS_VISIBLE |  
WS_TABSTOP,  
    136, 13, 60, 103
```

- **CONTROL** es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los combobox sólo sirve como información, y no se usa.
- id es el identificador del control. El identificador será necesario para inicializar y leer los contenidos y selecciones del combobox.
- class es la clase de control, en nuestro caso "COMBOBOX".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un **estilo combobox** y varios **de ventana**:
  - **CBS\_SORT**: Indica que los valores en la lista se aparecerán por orden alfabético. **CBS\_DROPDOWN** crea un ComboBox del tipo DropDown. **CBS\_DROPDOWNLIST** crea un ComboBox del tipo DropDownList.
  - **WS\_CHILD**: crea el control como una ventana hija.
  - **WS\_VISIBLE**: crea una ventana inicialmente visible.
  - **WS\_TABSTOP**: para que cuando el foco cambie de control al pulsar TAB, pase por este control.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

# Iniciar controles ComboBox

Iniciar los controles ComboBox es análogo a iniciar los controles ListBox. En general, necesitaremos introducir una lista de valores en el listBox, para que el usuario pueda usarla.

El lugar adecuado para hacerlo también es al procesar el mensaje `WM_INITDIALOG` de nuestro cuadro de diálogo, y el mensaje para añadir cadenas es `CB_ADDSTRING`. Hay que recordar también que para enviar mensajes a un control se usa la función `SendDlgItemMessage`.

Para hacer más fácil la inicialización de las listas, usaremos los mismos valores en las tres. Para ello definiremos, dentro de nuestra estructura de datos para compartir con el cuadro de diálogo, un array de cadenas con los valores necesario para inicializar los ComboBox:

También usaremos la misma estructura para almacenar los valores iniciales y de la última selección de los tres comboboxes:

```
/* Datos de la aplicación */
typedef struct stDatos {
    char Lista[6][80]; // Valores de los comboboxes
    char Item[3][80]; // Opciones elegidas
} DATOS;
```

Y asignaremos valores iniciales a las selecciones y a la lista de opciones al procesar el mensaje `WM_CREATE`:

```
static DATOS Datos;
int i;
...
case WM_CREATE:
    hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
    strcpy(Datos.Item[0], "a");
    strcpy(Datos.Item[1], "c");
    strcpy(Datos.Item[2], "e");
```

```

        for(i = 0; i < 6; i++)
            sprintf(Datos.Lista[i], "%c) Opción %c",
'a'+i, 'A'+i);
        return 0;

```

La parte de inicialización de los comboboxes se hace al procesar el mensaje **WM\_INITDIALOG**:

```

int i;
static DATOS *Datos;
...
case WM_INITDIALOG:
    Datos = (DATOS*)lParam;
    // Añadir cadenas. Mensaje: LB_ADDSTRING
    for(i = 0; i < 6; i++) {
        SendDlgItemMessage(hDlg, ID_COMBOBOX1,
            CB_ADDSTRING, 0, (LPARAM)Datos->Lista[i]);
        SendDlgItemMessage(hDlg, ID_COMBOBOX2,
            CB_ADDSTRING, 0, (LPARAM)Datos->Lista[i]);
        SendDlgItemMessage(hDlg, ID_COMBOBOX3,
            CB_ADDSTRING, 0, (LPARAM)Datos->Lista[i]);
    }
    SendDlgItemMessage(hDlg, ID_COMBOBOX1,
CB_SELECTSTRING,
        (LPARAM)-1, (LPARAM)Datos->Item[0]);
    SendDlgItemMessage(hDlg, ID_COMBOBOX2,
CB_SELECTSTRING,
        (LPARAM)-1, (LPARAM)Datos->Item[1]);
    SendDlgItemMessage(hDlg, ID_COMBOBOX3,
CB_SELECTSTRING,
        (LPARAM)-1, (LPARAM)Datos->Item[2]);
    SetFocus(GetDlgItem(hDlg, ID_COMBOBOX1));
    return FALSE;

```

También podemos preseleccionar alguna de las cadenas de los comboboxes. Para seleccionar una de las cadenas se usa un mensaje: **CB\_SELECTSTRING**. Usaremos el valor -1 en wParam para indicar que se busque en toda la lista.

## Devolver valores a la aplicación



```

// En el ComboList DropDownList usaremos:
indice = SendDlgItemMessage(hDlg,
ID_COMBOBOX3,
    CB_GETCURSEL, 0, 0);
SendDlgItemMessage(hDlg, ID_COMBOBOX3,
    CB_GETLBTEXT, indice, (LPARAM) Datos-
>Item[2]);
wsprintf(resultado, "%s\n%s\n%s",
    Datos-Item[0], Datos-Item[1], Datos-
>Item[2]);
MessageBox(hDlg, resultado, "Leido",
MB_OK);
EndDialog(hDlg, FALSE);
return TRUE;
case IDCANCEL:
EndDialog(hDlg, FALSE);
return FALSE;
}
break;
}

```

Pero ahora surge un problema. Si en un combobox introducimos una cadena que no está en nuestra lista, y posteriormente volvemos a entrar en el cuadro de diálogo, no podremos editar el valor inicial. Es más, ni siquiera nos será mostrado.

Para evitar eso deberíamos añadir cada nuevo valor introducido a la lista. Nuestro ejemplo es un poco limitado, ya que no tiene previsto que la lista pueda crecer, y desde luego, no guardará los valores de la lista cuando el programa termine, de modo que estén disponibles en sucesivas ejecuciones. Pero de momento nos conformaremos con ciertas modificaciones mínimas que ilustren cómo solventar este error.

Para empezar, reservaremos espacio suficiente para almacenar algunos valores extra, y modificaremos la estructura de datos:

```

#define MAX_CADENAS 100
...

/* Datos de la aplicación */
typedef struct stDatos {

```

```
int nCadenas;
char Lista[MAX_CADENAS][80];
char Item[3][80];
} DATOS;
```

También deberemos inicializar el valor de nCadenas, al procesar el mensaje [WM\\_CREATE](#):

```
Datos.nCadenas = 6;
```

Modificaremos la rutina para inicializar los ComboBoxes:

```
// Añadir cadenas. Mensaje: LB_ADDSTRING
for(i = 0; i < Datos->nCadenas; i++) {
    SendDlgItemMessage(hDlg, ID_COMBOBOX1,
        CB_ADDSTRING, 0, (LPARAM)Datos->Lista[i]);
    SendDlgItemMessage(hDlg, ID_COMBOBOX2,
        CB_ADDSTRING, 0, (LPARAM)Datos->Lista[i]);
    SendDlgItemMessage(hDlg, ID_COMBOBOX3,
        CB_ADDSTRING, 0, (LPARAM)Datos->Lista[i]);
}

...
```

Y para leer los valores introducidos, y añadirlos a la lista si no están. Para eso usaremos el mensaje [CB\\_FINDSTRINGEXACT](#), que buscará una cadena entre los valores almacenados en la lista, si se encuentra devolverá un índice, y si no, el valor [CB\\_ERR](#).

Existe otro mensaje parecido, [CB\\_FINDSTRING](#), pero no nos vale, porque localizará la primera cadena de la lista que comience con los mismos caracteres que la cadena que buscamos. Por ejemplo, si hay un valor en la lista "valor a", y nosotros buscamos "valor", obtendremos el índice de la cadena "valor a", que no es lo que queremos, al menos en este ejemplo.

El proceso del comando [IDOK](#) quedaría así:



```

        case IDOK:
            // En el ComboList Simple usaremos:
            GetDlgItemText(hDlg, ID_COMBOBOX1, Datos-
>Item[0], 80);
            if(SendDlgItemMessage(hDlg, ID_COMBOBOX1,
CB_FINDSTRINGEXACT,
                (WPARAM)-1, (LPARAM)Datos->Item[0]) ==
CB_ERR)
                strcpy(Datos->Lista[Datos->nCadenas++],
Datos->Item[0]);
            // En el ComboList DropDown usaremos:
            SendDlgItemMessage(hDlg, ID_COMBOBOX2,
WM_GETTEXT,
                80, (LPARAM)Datos->Item[1]);
            if(SendDlgItemMessage(hDlg, ID_COMBOBOX1,
CB_FINDSTRINGEXACT,
                (WPARAM)-1, (LPARAM)Datos->Item[1]) ==
CB_ERR &&
                strcmp(Datos->Item[0], Datos->Item[1]))
                strcpy(Datos->Lista[Datos->nCadenas++],
Datos->Item[1]);
            // En el ComboList DropDownList usaremos:
            indice = SendDlgItemMessage(hDlg,
ID_COMBOBOX3,
                CB_GETCURSEL, 0, 0);
            SendDlgItemMessage(hDlg, ID_COMBOBOX3,
                CB_GETLBTEXT, indice, (LPARAM)Datos-
>Item[2]);
            wsprintf(resultado, "%s\n%s\n%s",
                Datos->Item[0], Datos->Item[1], Datos-
>Item[2]);
            MessageBox(hDlg, resultado, "Leido",
MB_OK);
            EndDialog(hDlg, FALSE);
            return TRUE;

```

## Ejemplo 10

# Capítulo 12 Control básico Scrollbar

Veremos ahora el siguiente control básico: la barra de desplazamiento o Scrollbar.

Las ventanas pueden mostrar contenidos que ocupan más espacio del que cabe en su interior, cuando eso sucede se suelen agregar unos controles en forma de barra que permiten desplazar el contenido a través del área de la ventana de modo que el usuario pueda ver las partes ocultas del documento.

Pero las barras de scroll pueden usarse para introducir otros tipos de datos en nuestras aplicaciones, en general, cualquier magnitud de la que sepamos el máximo y el mínimo, y que tenga un rango valores finito. Por ejemplo un control de volumen, de 0 a 10, o un termostato de -15° a 60°.

Las barras de desplazamiento tienen varias partes o zonas diferenciadas, cada una con su función particular. Me imagino que ya las conoces, pero las veremos desde el punto de vista de un programador.

Una barra de desplazamiento consiste en un rectángulo sombreado con un botón de flecha en cada extremo, y una caja en el interior del rectángulo (llamado normalmente thumb). La barra de desplazamiento representa la longitud o anchura completa del documento, y la caja interior la porción visible del documento dentro de la ventana. La posición de la caja cambia cada vez que el usuario desplaza el documento para ver diferentes partes de él. También se modifica el tamaño de la caja para adaptarlo a la proporción del documento que es visible. Cuanta más porción del documento resulte visible, mayor será el tamaño de la caja, y viceversa.

Hay dos modalidades de ScrollBars: horizontales y verticales.

El usuario puede desplazar el contenido de la ventana pulsando uno de los botones de flecha, pulsando en la zona sombreada no ocupada por el thumb, o desplazando el propio thumb. En el primer caso se desplazará el equivalente a una unidad (si es texto, una línea o columna). En el segundo, el contenido se desplazará en la porción equivalente al contenido de una ventana. En el tercer caso, la cantidad de documento desplazado dependerá de la distancia que se desplace el thumb.

Hay que distinguir los controles ScrollBar de las barras de desplazamiento estándar. Aparentemente son iguales, y se comportan igual, los primeros están en el área de cliente de la ventana, pero las segundas no, éstas se crean y se muestran junto con la ventana. Para añadir estas barras a tu ventana, basta con crearla con los estilos `WS_HSCROLL`, `WS_VSCROLL` o ambos. `WS_HSCROLL` añade una barra horizontal y `WS_VSCROLL` una vertical.

Un control scroll bar es una ventana de control de la clase `SCROLLBAR`. Se pueden crear tantas barras de scroll como se quiera, pero el programador es el encargado de especificar el tamaño y la posición de la barra.

## Ficheros de recursos

Para nuestro ejemplo incluiremos un control ScrollBar de cada tipo, aunque en realidad son casi idénticos en comportamiento:

```
#include <windows.h>
#include "win011.h"

Menu MENU
BEGIN
    POPUP "&Principal"
    BEGIN
        MENUITEM "&Diálogo", CM_DIALOGO
    END
END
```

```

DialogoPrueba DIALOG 0, 0, 189, 106
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
CAPTION "Scroll bars"
FONT 8, "Helv"
BEGIN
    CONTROL "ScrollBar1", ID_SCROLLH, "SCROLLBAR",
        SBS_HORZ | WS_CHILD | WS_VISIBLE,
        7, 3, 172, 9
    CONTROL "Scroll 1:", -1, "STATIC",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        24, 18, 32, 8
    CONTROL "Edit1", ID_EDITH, "EDIT",
        ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_TABSTOP,
        57, 15, 32, 12
    CONTROL "ScrollBar2", ID_SCROLLV, "SCROLLBAR",
        SBS_VERT | WS_CHILD | WS_VISIBLE,
        7, 15, 9, 86
    CONTROL "Scroll 2:", -1, "STATIC",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        23, 41, 32, 8
    CONTROL "Edit2", ID_EDITV, "EDIT",
        ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_TABSTOP,
        23, 51, 32, 12
    CONTROL "Aceptar", IDOK, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        40, 87, 50, 14
END

```

Para ver cómo funcionan las barras de scroll hemos añadido dos controles Edit, que mostrarán los valores seleccionados en cada control ScrollBar. Para más detalles acerca de estos controles ver [control scrollbar](#).

Ahora veremos más cosas sobre los estilos de los controles ScrollBar:

```

CONTROL "ScrollBar1", ID_SCROLLH, "SCROLLBAR",
    SBS_HORZ | WS_CHILD | WS_VISIBLE,
    7, 3, 172, 9

```

```
CONTROL "ScrollBar2", ID_SCROLLV, "SCROLLBAR",  
    SBS_VERT | WS_CHILD | WS_VISIBLE,  
    7, 15, 9, 86
```

- **CONTROL** es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los scrollbar sólo sirve como información, y no se usa.
- id es el identificador del control. El identificador será necesario para inicializar y leer el valor del scrollbar, así como para manipular los mensajes que produzca.
- class es la clase de control, en nuestro caso "SCROLLBAR".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un **estilo scrollbar** y varios **de ventana**:
  - **SBS\_HORZ**: Indica se trata de un scrollbar horizontal.
  - **SBS\_VERT**: Indica se trata de un scrollbar vertical.
  - **WS\_CHILD**: crea el control como una ventana hija.
  - **WS\_VISIBLE**: crea una ventana inicialmente visible.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

## Iniciar controles Scrollbar

Los controles Scrollbar tienen varios tipos de parámetros que hay que iniciar. Los límites de valores mínimo y máximo, y también el valor actual.

El lugar adecuado para hacerlo también es al procesar el mensaje **WM\_INITDIALOG** de nuestro cuadro de diálogo, y para ajustar los parámetros podemos usar mensajes o funciones. En el caso de hacerlo con mensajes hay que usar la función **SendDlgItemMessage**.

También usaremos una estructura para almacenar los valores iniciales y de la última selección de las dos barras de desplazamiento:

```
// Datos de la aplicación
typedef struct stDatos {
    int ValorH;
    int ValorV;
} DATOS;
```

Declararemos los datos como estáticos en el procedimiento de ventana, y asignaremos valores iniciales a los controles al procesar el mensaje [WM\\_CREATE](#):

```
static DATOS Datos;
...
case WM_CREATE:
    hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
    Datos.ValorH = 10;
    Datos.ValorV = 32;
    return 0;
```

Pasaremos un puntero a nuestra estructura de datos al procedimiento de diálogo usando el parámetros lParam y la función [DialogBoxParam](#):

```
DialogBoxParam(hInstance, "DialogoPrueba", hwnd, DlgProc,
(LPARAM) &Datos);
```

En el procedimiento de diálogo disponemos de un puntero estático a la estructura de datos, que inicializaremos al procesar el mensaje [WM\\_INITDIALOG](#).

La parte de inicialización de los scrollbars es como sigue. Hemos inicializado el scrollbar horizontal usando las funciones y el vertical usando los mensajes:

```
static DATOS *Datos;
```

```

...
    case WM_INITDIALOG:
        Datos = (DATOS*)lParam;
        SetScrollRange(GetDlgItem(hDlg, ID_SCROLLH),
SB_CTL,
        0, 100, TRUE);
        SetScrollPos(GetDlgItem(hDlg, ID_SCROLLH),
SB_CTL,
        Datos->ValorH, TRUE);
        SetDlgItemInt(hDlg, ID_EDITH, (UINT)Datos-
>ValorH, FALSE);
        SendDlgItemMessage(hDlg, ID_SCROLLV,
SBM_SETRANGE,
        (WPARAM)0, (LPARAM)50);
        SendDlgItemMessage(hDlg, ID_SCROLLV, SBM_SETPOS,
        (WPARAM)Datos->ValorV, (LPARAM)TRUE);
        SetDlgItemInt(hDlg, ID_EDITV, (UINT)Datos-
>ValorV, FALSE);

```

Para iniciar el rango de valores del scrollbar se usa la función [SetScrollRange](#) o el mensaje [SBM\\_SETRANGE](#). Para cambiar el valor seleccionado o posición se usa la función [SetScrollPos](#) o el mensaje [SBM\\_SETPOS](#).

## Iniciar controles scrollbar: estructura SCROLLINFO

A partir de la versión 4.0 de Windows existe otro mecanismo para inicializar los scrollbars. También tiene la doble forma de mensaje y función. Su uso se recomienda en lugar de [SetScrollRange](#), que sólo se conserva por compatibilidad con Windows 3.x.

Se trata de la función [SetScrollInfo](#) y del mensaje [SBM\\_SETSCROLLINFO](#). También es necesaria una estructura que se usará para pasar los parámetros tanto a la función como al mensaje: [SCROLLINFO](#).

Usando esta forma, el ejemplo anterior quedaría así:

```

static DATOS *Datos;
SCROLLINFO sih = {
    sizeof(SCROLLINFO),
    SIF_POS | SIF_RANGE | SIF_PAGE,
    0, 104,
    5,
    0,
    0;
SCROLLINFO siv = {
    sizeof(SCROLLINFO),
    SIF_POS | SIF_RANGE | SIF_PAGE,
    0, 54,
    5,
    0,
    0;

...
case WM_INITDIALOG:
    Datos = (DATOS*)lParam;
    sih.nPos = Datos->ValorH;
    siv.nPos = Datos->ValorV;
    SetScrollInfo(GetDlgItem(hDlg, ID_SCROLLH),
SB_CTL, &sih, TRUE);
    SetDlgItemInt(hDlg, ID_EDITH, (UINT)Datos-
>ValorH, FALSE);
    SendDlgItemMessage(hDlg, ID_SCROLLV,
SBM_SETSCROLLINFO,
                                (WPARAM)TRUE, (LPARAM)&siv);
    SetDlgItemInt(hDlg, ID_EDITV, (UINT)Datos-
>ValorV, FALSE);
    return FALSE;

```

El segundo campo de la estructura **SCROLLINFO** consiste en varios bits que indican qué parámetros de la estructura se usarán para inicializar los scrollbars. Hemos incluido la posición, el rango y el valor de la página. Sería equivalente haber puesto únicamente **SIF\_ALL**.

El valor de la página no lo incluíamos antes, y veremos que será útil al procesar los mensajes que provienen de los controles scrollbar. Además el tamaño de la caja de desplazamiento se ajusta de modo que esté a escala en relación con el tamaño total del



control scrollbar. Si hubiéramos definido una página de 50 y un rango de 0 a 100, el tamaño de la caja sería exactamente la mitad del tamaño del scrollbar.

Hay que tener en cuenta que el valor máximo que podremos establecer en un control no es siempre el que nosotros indicamos en el miembro `nMax` de la estructura `SCROLLINFO`. Este valor depende del valor de la página (`nPage`), y será `nMax-nPage+1`. Así que si queremos que nuestro control pueda devolver 100, y la página tiene un valor de 5, debemos definir `nMax` como 104. Este funcionamiento está diseñado para scrollbars como los que incluyen las ventanas, donde la caja indica la porción del documento que se muestra en su interior.

## Procesar los mensajes procedentes de controles Scrollbar

Cada vez que el usuario realiza alguna acción en un control Scrollbar se envía un mensaje `WM_HSCROLL` o `WM_VSCROLL`, dependiendo del tipo de control, a la ventana donde está insertado. En realidad pasa algo análogo con todos los controles, pero en el caso de los Scrollbars, es imprescindible que el programa procese algunos de esos mensajes adecuadamente.

Estos mensajes entregan distintos valores en la palabra de menor peso del parámetro `wParam`, según la acción del usuario sobre el control. En la palabra de mayor peso se incluye la posición actual y en `lParam` el manipulador de ventana del control.

De modo que nuestra rutina para manejar los mensajes de los scrollbars debe ser capaz de distinguir el control del que procede el mensaje y el tipo de acción, para actuar en consecuencia.

En nuestro caso es irrelevante la orientación de la barra de scroll, podemos distinguirlos por el identificador de ventana, de todos modos procesaremos cada uno de los dos mensajes con una rutina distinta.

Para no recargar en exceso el procedimiento de ventana del diálogo, crearemos una función para procesar los mensajes de las barras de scroll. Y la llamaremos al recibir esos mensajes:

```
switch (msg)                                /* manipulador del mensaje
*/
{
...
    case WM_HSCROLL:
        ProcesarScrollH(hDlg, (HWND)lParam,
            (int)LOWORD(wParam), (int)HIWORD(wParam));
        return FALSE;
    case WM_VSCROLL:
        ProcesarScrollV(hDlg, (HWND)lParam,
            (int)LOWORD(wParam), (int)HIWORD(wParam));
        return FALSE;
...
}
```

Los códigos que tenemos que procesar son los siguientes:

- **SB\_BOTTOM**: desplazamiento hasta el principio de la barra, en verticales arriba y en horizontales a la izquierda.
- **SB\_TOP**: desplazamiento hasta el final de la barra, en verticales abajo y en horizontales a la derecha.
- **SB\_LINERIGHT** y **SB\_LINEDOWN**: desplazamiento una línea a la derecha en horizontales o abajo en verticales.
- **SB\_LINELEFT** y **SB\_LINEUP**: desplazamiento una línea a la izquierda en horizontales o arriba en verticales.
- **SB\_PAGERIGHT** y **SB\_PAGEDOWN**: desplazamiento de una página a la derecha en horizontales y abajo en verticales.
- **SB\_PAGELEFT** y **SB\_PAGEUP**: desplazamiento un párrafo a la izquierda en horizontales o arriba en verticales.
- **SB\_THUMBPOSITION**: se envía cuando el thumb está en su posición final.
- **SB\_THUMBTRACK**: el thumb se está moviendo.
- **SB\_ENDSCROLL**: el usuario ha liberado el thumb en una nueva posición.

En nuestro caso, no haremos que la variable asociada se actualice hasta que pulsemos el botón de aceptar, pero actualizaremos la posición del thumb y el valor del control edit asociado a cada scrollbar.

Veamos por ejemplo la rutina para tratar el scroll horizontal:

```
void ProcesarScrollH(HWND hDlg, HWND Control, intCodigo,
int Posicion)
{
    int Pos = GetScrollPos(Control, SB_CTL);

    switch(Codigo) {
        case SB_BOTTOM:
            Pos = 0;
            break;
        case SB_TOP:
            Pos = 100;
            break;
        case SB_LINERIGHT:
            Pos++;
            break;
        case SB_LINELEFT:
            Pos--;
            break;
        case SB_PAGERIGHT:
            Pos += 5;
            break;
        case SB_PAGELEFT:
            Pos -= 5;
            break;
        case SB_THUMBPOSITION:
        case SB_THUMBTRACK:
            Pos = Posicion;
        case SB_ENDSCROLL:
            break;
    }
    if(Pos < 0) Pos = 0;
    if(Pos > 100) Pos = 100;
    SetDlgItemInt(hDlg, ID_EDITH, (UINT)Pos, FALSE);
    SetScrollPos(Control, SB_CTL, Pos, TRUE);
}
```

Como puede observarse, actualizamos el valor de la posición dependiendo del código recibido. Nos aseguramos de que está dentro de los márgenes permitidos y finalmente actualizamos el contenido del control edit. La función para el scrollbar vertical es análoga, pero cambiando los identificadores de los códigos y los valores de los límites.

Hemos usado una función nueva, [GetScrollPos](#) para leer la posición actual del thumb.

## Procesar mensajes de scrollbar usando SCROLLINFO

Como comentamos antes, a partir de la versión 4.0 de Windows existe otro mecanismo para inicializar los scrollbars. También tiene la doble forma de mensaje y función. Su uso se recomienda en lugar de [GetScrollRange](#), que sólo se conserva por compatibilidad con Windows 3.x.

Usando [GetScrollInfo](#), la función para procesar el mensaje del scrollbar horizontal quedaría así:

```
void ProcesarScrollH(HWND hDlg, HWND Control, intCodigo,
int Posicion)
{
    SCROLLINFO si = {
        sizeof(SCROLLINFO),
        SIF_ALL, 0, 0, 0, 0, 0};

    GetScrollInfo(Control, SB_CTL, &si);

    switch(Codigo) {
        case SB_BOTTOM:
            si.nPos = si.nMin;
            break;
        case SB_TOP:
            si.nPos = si.nMax;
            break;
        case SB_LINEDOWN:
            si.nPos++;
    }
```

```

        break;
    case SB_LINEUP:
        si.nPos--;
        break;
    case SB_PAGEDOWN:
        si.nPos += si.nPage;
        break;
    case SB_PAGEUP:
        si.nPos -= si.nPage;
        break;
    case SB_THUMBPOSITION:
    case SB_THUMBTRACK:
        si.nPos = Posicion;
    case SB_ENDSCROLL:
        break;
    }
    if(si.nPos < si.nMin) si.nPos = si.nMin;
    if(si.nPos > si.nMax-si.nPage+1) si.nPos = si.nMax-
si.nPage+1;

    SetScrollInfo(Control, SB_CTL, &si, true);
    SetDlgItemInt(hDlg, ID_EDITH, (UINT)si.nPos, FALSE);
}

```

Usamos los valores de la estructura para acotar la posición de la caja y para avanzar y retroceder de página, esto hace que nuestra función sea más independiente y que use menos constantes definidas en el fuente.

También se puede usar el mensaje [SBM\\_GETSCROLLINFO](#), basta con cambiar la línea:

```
GetScrollInfo(Control, SB_CTL, &si);
```

por esta otra:

```
SendDlgItemMessage(hDlg, ID_SCROLLH,
    SBM_GETSCROLLINFO, 0, (LPARAM)&siv);
```

## Devolver valores a la aplicación

Cuando el usuario ha decidido que los valores son los adecuados pulsará el botón de Aceptar. En ese momento deberemos capturar los valores de los controles scroll y actualizar la estructura de parámetros.

También en este caso podemos usar un mensaje o una función para leer la posición del thumb del scrollbar. La función ya la hemos visto un poco más arriba, se trata de [GetScrollPos](#). El mensaje es [SBM\\_GETPOS](#), ambos devuelven el valor de la posición actual del thumb del control.

Usaremos los dos métodos, uno con cada control. El lugar adecuado para leer esos valores sigue siendo el tratamiento del mensaje [WM\\_COMMAND](#):

```
case WM_COMMAND:
    switch(LOWORD(wParam)) {
        case IDOK:
            Datos->ValorH =
GetScrollPos(GetDlgItem(hDlg, ID_SCROLLH), SB_CTL);
            Datos->ValorV = SendDlgItemMessage(hDlg,
ID_SCROLLV,
                SBM_GETPOS, 0, 0);
            EndDialog(hDlg, FALSE);
            return TRUE;
        case IDCANCEL:
            EndDialog(hDlg, FALSE);
            return FALSE;
```

## Ejemplos 11 y 12

# Capítulo 13 Control básico

## Groupbox

Los GroupBoxes son un estilo de botón que se usa para agrupar controles. Generalmente se usan con controles RadioButton, pero se pueden agrupar controles de cualquier tipo.

El comportamiento es puramente estático, es decir, actúan sólo como marcas y facilitan al usuario el acceso a distintos grupos de controles asociados en función de alguna propiedad común.

## Ficheros de recursos

Para comprobar cómo funcionan los groupboxes agruparemos dos conjuntos de controles edit:

```
#include <windows.h>
#include "win013.h"

Menu MENU
BEGIN
    POPUP "&Principal"
    BEGIN
        MENUITEM "&Diálogo", CM_DIALOGO
    END
END

DialogoPrueba DIALOG 0, 0, 145, 87
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
CAPTION "Group boxes"
FONT 8, "Helv"
BEGIN
    CONTROL "Grupo &1", ID_GROUPBOX1, "BUTTON",
        BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP,
        8, 4, 64, 65
```

```

CONTROL "Botón 1", ID_BOTON1, "BUTTON",
    BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
    16, 17, 50, 14
CONTROL "Botón 2", ID_BOTON2, "BUTTON",
    BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
    16, 33, 50, 14
CONTROL "Botón 3", ID_BOTON3, "BUTTON",
    BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
    16, 49, 50, 14
CONTROL "Grupo &2", ID_GROUPBOX2, "BUTTON",
    BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP,
    74, 4, 66, 65
CONTROL "Botón 4", ID_BOTON4, "BUTTON",
    BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
    81, 17, 50, 14
CONTROL "Botón 5", ID_BOTON5, "BUTTON",
    BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
    81, 33, 50, 14
CONTROL "Botón 6", ID_BOTON6, "BUTTON",
    BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
    81, 49, 50, 14
CONTROL "Aceptar", IDOK, "BUTTON",
    BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_GROUP | WS_TABSTOP,
    16, 72, 50, 14
CONTROL "Cancelar", IDCANCEL, "BUTTON",
    BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
    80, 72, 50, 14
END

```

Hemos añadido los nuevos controles GroupBox. Para más detalles acerca de estos controles ver [controles button](#).

Como se puede observar, un Groupbox no es más que un botón con el estilo **BS\_GROUPBOX**, la principal propiedad de los controles agrupados bajo un groupbox es que es posible moverse a través de ellos usando las teclas del cursor.



```
CONTROL "Grupo &1", ID_GROUPBOX1, "button",  
    BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP,  
    8, 4, 64, 65
```

- **CONTROL** es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los groupbox será el texto que aparecerá en la esquina superior izquierda del control, y que servirá para identificar el grupo.
- id es el identificador del control. El identificador será necesario en algunos casos para vincular los controles entre sí, le veremos más adelante cuando estudiemos los Radio Buttons.
- class es la clase de control, en nuestro caso "BUTTON".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un **estilo button** y varios **de ventana**:
  - **BS\_GROUPBOX**: Indica que se trata de un botón con el estilo GroupBox.
  - **WS\_CHILD**: crea el control como una ventana hija.
  - **WS\_VISIBLE**: crea una ventana inicialmente visible.
  - **WS\_GROUP**: marca el control como comienzo de un grupo, el grupo termina cuando empieza el grupo siguiente o terminen los controles.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

## Iniciar controles GroupBox

Los controles GroupBox no precisan inicialización.

## Devolver valores a la aplicación

Tampoco hay ningún valor que retornar desde un control GroupBox.

## **Ejemplo 13**

# Capítulo 14 Control básico

## Checkbox

En realidad no se trata más que de otro estilo de botón.

Normalmente, los CheckBoxes pueden tomar dos valores, encendido y apagado. Aunque también existen Checkbox de tres estados, en ese caso, el tercer estado corresponde al de inhibido. Los CheckBoxes se usan típicamente para leer opciones que sólo tienen dos posibilidades, del tipo cuya respuesta es sí o no, encendido o apagado, verdadero o falso, etc.

Aunque a menudo se agrupan, en realidad los checkboxes son independientes, cada uno suele tomar un valor de tipo booleano, independientemente de los valores del resto del grupo, si es que pertenece a uno.

El aspecto normal es el de una pequeña caja cuadrada con un texto a uno de los lados, normalmente a la derecha. Cuando está activo se muestra una marca en el interior de la caja, cuando no lo está, la caja aparece vacía. También es posible mostrar el checkbox como un botón corriente, en ese caso, al activarse se quedará pulsado.

## Ficheros de recursos

Vamos a mostrar algunos de los posibles aspectos de los CheckBoxes:

```
#include <windows.h>
#include "win014.h"
```

```
Menu MENU
```

```

BEGIN
    POPUP "&Principal"
    BEGIN
        MENUITEM "&Diálogo", CM_DIALOGO
    END
END

DialogoPrueba DIALOG 0, 0, 168, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
CAPTION "CheckBoxes"
FONT 8, "Helv"
BEGIN
    CONTROL "Normal", ID_NORMAL, "BUTTON",
        BS_CHECKBOX | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        12, 4, 60, 12
    CONTROL "Auto", ID_AUTO, "BUTTON",
        BS_AUTOCHECKBOX | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        84, 4, 72, 12
    CONTROL "Tres estados", ID_TRISTATE, "BUTTON",
        BS_3STATE | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        12, 17, 60, 12
    CONTROL "Auto tres estados", ID_AUTOTRISTATE, "BUTTON",
        BS_AUTO3STATE | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        84, 16, 76, 12
    CONTROL "Auto Push", ID_AUTOPUSH, "BUTTON",
        BS_AUTOCHECKBOX | BS_PUSHLIKE | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        12, 32, 60, 12
    CONTROL "Auto Tristate Push", ID_AUTOTRIPUSH, "BUTTON",
        BS_AUTO3STATE | BS_PUSHLIKE | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        83, 32, 75, 12
    CONTROL "Derecha", ID_DERECHA, "BUTTON",
        BS_AUTOCHECKBOX | BS_LEFTTEXT | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        12, 50, 60, 12
    CONTROL "Plano", ID_PLANO, "BUTTON",
        BS_AUTOCHECKBOX | BS_FLAT | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        84, 50, 60, 12
    CONTROL "Aceptar", IDOK, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        12, 69, 50, 14
    CONTROL "Cancelar", IDCANCEL, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,

```

```
84, 69, 50, 14  
END
```

Para ver más detalles acerca de este tipo de controles ver [controles button](#).

Como se puede observar, un CheckBox es un tipo de botón con uno de los siguientes estilos [BS\\_CHECKBOX](#), [BS\\_AUTOCHECKBOX](#), [BS\\_3STATE](#) o [BS\\_AUTO3STATE](#).

Podemos dividir los CheckBoxes en cuatro categorías diferentes, dependiendo de si son o no automáticos o de si son de dos o tres estados.

```
CONTROL "Normal", ID_NORMAL, "BUTTON",  
    BS_CHECKBOX | WS_CHILD | WS_VISIBLE | WS_TABSTOP,  
    12, 4, 60, 12
```

- [CONTROL](#) es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los CheckBox será el texto que aparecerá acompañando a la caja o en el interior del botón y que servirá para identificar el valor a editar.
- id es el identificador del control. El identificador será necesario en algunos casos para procesar los comandos procedentes del CheckBox, en el caso de los no automáticos será la única forma de tratarlos.
- class es la clase de control, en nuestro caso "BUTTON".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un [estilo button](#) y varios [de ventana](#):
  - [BS\\_CHECKBOX](#): Indica que se trata de un CheckBox de dos estados.
  - [BS\\_AUTOCHECKBOX](#): Indica que se trata de un CheckBox de dos estados automático.
  - [BS\\_3STATE](#): Indica que se trata de un CheckBox de tres estados.

- **BS\_AUTO3STATE**: Indica que se trata de un CheckBox de tres estados automático.
- **BS\_PUSHLIKE**: Indica que se trata de un CheckBox con la apariencia de un botón corriente.
- **BS\_LEFTTEXT**: Indica que el texto del CheckBox se sitúa a la izquierda de la caja.
- **BS\_FLAT**: Indica apariencia plana, sin las sombras que simulan 3D.
- **WS\_CHILD**: crea el control como una ventana hija.
- **WS\_VISIBLE**: crea una ventana inicialmente visible.
- **WS\_TABSTOP**: para que cuando el foco cambie de control al pulsar TAB, pase por este control.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

## Iniciar controles CheckBox

Los controles CheckBox suelen precisar inicialización.

Para este ejemplo también usaremos variables globales para almacenar los valores de las variables que se pueden editar con los CheckBoxes.

```
// Datos de la aplicación
typedef struct stDatos {
    BOOL Normal;
    BOOL Auto;
    int TriState;
    int AutoTriState;
    BOOL AutoPush;
    int AutoTriPush;
    BOOL Derecha;
    BOOL Plano;
} DATOS;
```

Creamos una variable estática en nuestro procedimiento de ventana para almacenar los datos, y le daremos valores iniciales a las variables de la aplicación, al procesar el mensaje [WM\\_CREATE](#):

```
static DATOS Datos;
static HINSTANCE hInstance;
...
case WM_CREATE:
    hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
    /* Inicialización */
    Datos.Normal = TRUE;
    Datos.Auto = TRUE;
    Datos.TriState = (int)FALSE;
    Datos.AutoTriState = (int)FALSE;
    Datos.AutoPush = FALSE;
    Datos.AutoTriPush = (int)TRUE;
    Datos.Derecha = TRUE;
    Datos.Plano = FALSE;
    return 0;
```

Al crear el cuadro de diálogo, usaremos la función [DialogBoxParam](#), y en el parámetro lParam pasaremos el puntero a nuestra estructura de datos:

```
DialogBoxParam(hInstance, "DialogoPrueba",
    hwnd, DlgProc, (LPARAM)&Datos);
```

Como siempre, para establecer los valores iniciales de los controles CheckBox usaremos el mensaje [WM\\_INITDIALOG](#) del procedimiento de diálogo.

Para eso usaremos la función [CheckDlgButton](#) o el mensaje [BM\\_SETCHECK](#), en este último caso, emplearemos la función [SendDlgItemMessage](#).

De nuevo ilustraremos el ejemplo usando los dos métodos:

```
#define DESHABILITADO -1
```

```

... static DATOS *Datos;
...
    case WM_INITDIALOG:
        Datos = (DATOS*)lParam;
        // Estado inicial de los checkbox
        CheckDlgButton(hDlg, ID_NORMAL,
            Datos->Normal ? BST_CHECKED : BST_UNCHECKED);
        CheckDlgButton(hDlg, ID_AUTO,
            Datos->Auto ? BST_CHECKED : BST_UNCHECKED);
        if(Datos->TriState != DESHABILITADO)
            CheckDlgButton(hDlg, ID_TRISTATE,
                Datos->TriState ? BST_CHECKED :
BST_UNCHECKED);
        else
            CheckDlgButton(hDlg, ID_TRISTATE,
BST_INDETERMINATE);
        if(Datos->AutoTriState != DESHABILITADO)
            CheckDlgButton(hDlg, ID_AUTOTRISTATE,
                Datos->AutoTriState ? BST_CHECKED :
BST_UNCHECKED);
        else
            CheckDlgButton(hDlg, ID_AUTOTRISTATE,
BST_INDETERMINATE);
        CheckDlgButton(hDlg, ID_AUTOPUSH,
            Datos->AutoPush ? BST_CHECKED :
BST_UNCHECKED);
        // Usando mensajes:
        if(Datos->AutoTriPush != DESHABILITADO)
            SendDlgItemMessage(hDlg, ID_AUTOTRIPUSH,
BM_SETCHECK,
                Datos->AutoTriPush ? (LPARAM)BST_CHECKED :
(WPARAM)BST_UNCHECKED, 0);
        else
            SendDlgItemMessage(hDlg, ID_AUTOTRIPUSH,
BM_SETCHECK,
                (LPARAM)BST_INDETERMINATE, 0);
        SendDlgItemMessage(hDlg, ID_DERECHA, BM_SETCHECK,
            Datos->Derecha ? (LPARAM)BST_CHECKED :
(WPARAM)BST_UNCHECKED, 0);
        SendDlgItemMessage(hDlg, ID_PLANO, BM_SETCHECK,
            Datos->Plano ? (LPARAM)BST_CHECKED :
(WPARAM)BST_UNCHECKED, 0);
        SetFocus(GetDlgItem(hDlg, ID_NORMAL));
        return FALSE;

```



Con esto, el estado inicial de los CheckBoxes será correcto.

## Procesar mensajes de los CheckBox

En ciertos casos, será necesario procesar algunos de los mensajes procedentes de los CheckBoxes.

Concretamente, en el caso de los CheckBox no automáticos, su estado no cambia cuando el usuario actúa sobre ellos, sino que será el programa quien deba actualizar ese estado.

Para actualizar el estado de los CheckBoxes cada vez que el usuario actúe sobre ellos debemos procesar los mensajes `WM_COMMAND` procedentes de ellos.

Quizás, la primera intención sea modificar las variables almacenadas en la estructura Datos para adaptarlas a los nuevos valores. Pero recuerda que es posible que el usuario pulse el botón de "Cancelar". En ese caso, los valores de la estructura Datos no deberían cambiar.

Tenemos dos opciones. Una es usar variables auxiliares para almacenar el estado actual de los CheckBoxes. Otra es leer el estado de los controles directamente. Este último sistema es más seguro, ya que previene el que las variables auxiliares y los controles tengan valores diferentes.

Para leer el estado de los controles tenemos dos posibilidades, como siempre: usar la función `IsDlgButtonChecked` o el mensaje `BM_GETCHECK`.

```
case WM_COMMAND:
    switch (LOWORD(wParam)) {
        case ID_NORMAL:
            if (SendDlgItemMessage(hDlg, ID_NORMAL,
BM_GETCHECK, 0, 0) == BST_CHECKED)
                SendDlgItemMessage(hDlg, ID_NORMAL,
BM_SETCHECK,
                                (WPARAM)BST_UNCHECKED, 0);
            else
                SendDlgItemMessage(hDlg, ID_NORMAL,
```

```

BM_SETCHECK,
                (WPARAM)BST_CHECKED, 0);
    return TRUE;

    case ID_TRISTATE:
        if(IsDlgButtonChecked(hDlg, ID_TRISTATE) ==
BST_INDETERMINATE)
            CheckDlgButton(hDlg, ID_TRISTATE,
BST_UNCHECKED);
        else if(IsDlgButtonChecked(hDlg,
ID_TRISTATE) == BST_CHECKED)
            CheckDlgButton(hDlg, ID_TRISTATE,
BST_INDETERMINATE);
        else
            CheckDlgButton(hDlg, ID_TRISTATE,
BST_CHECKED);
    return TRUE;
}

```

En este ejemplo intentamos simular el comportamiento de los CheckBoxes automáticos, pero lo normal es que para eso se usen CheckBoxes automáticos. Los no automáticos pueden tener el comportamiento que nosotros prefiramos, en eso consiste su utilidad.

## Devolver valores a la aplicación

Por supuesto, lo normal también será que queramos retornar los valores actuales seleccionados en cada CheckBox. A veces también necesitaremos leer el estado de algún control CheckBox durante la ejecución, por ejemplo, cuando eso influye en el estado de otros controles.

Cuando sólo nos interese devolver valores antes de cerrar el diálogo, leeremos esos valores al procesar el mensaje `WM_COMMAND` para el botón de "Aceptar".

Ya sabemos los dos modos de obtener el estado de los controles CheckBox, la función `IsDlgButtonChecked` y el mensaje `BM_GETCHECK`. Ahora sólo tenemos que añadir la lectura de ese

estado al procesamiento del mensaje **WM\_COMMAND** del botón "Aceptar".

```
        case IDOK:
            Datos->Normal = (IsDlgButtonChecked(hDlg,
ID_NORMAL) == BST_CHECKED);
            Datos->Auto = (IsDlgButtonChecked(hDlg,
ID_AUTO) == BST_CHECKED);
            if(IsDlgButtonChecked(hDlg, ID_TRISTATE) ==
BST_INDETERMINATE)
                Datos->TriState = DESHABILITADO;
            else
                Datos->TriState =
(IsDlgButtonChecked(hDlg, ID_TRISTATE) == BST_CHECKED);
                if(IsDlgButtonChecked(hDlg,
ID_AUTOTRISTATE) == BST_INDETERMINATE)
                    Datos->AutoTriState = DESHABILITADO;
                else
                    Datos->AutoTriState =
(IsDlgButtonChecked(hDlg, ID_AUTOTRISTATE) == BST_CHECKED);
                    Datos->AutoPush = (IsDlgButtonChecked(hDlg,
ID_AUTOPUSH) == BST_CHECKED);
                    if(IsDlgButtonChecked(hDlg, ID_AUTOTRIPUSH)
== BST_INDETERMINATE)
                        Datos->AutoTriPush = DESHABILITADO;
                    else
                        Datos->AutoTriPush =
(IsDlgButtonChecked(hDlg, ID_AUTOTRIPUSH) == BST_CHECKED);
                        Datos->Derecha = (IsDlgButtonChecked(hDlg,
ID_DERECHA) == BST_CHECKED);
                        Datos->Plano = (IsDlgButtonChecked(hDlg,
ID_PLANO) == BST_CHECKED);
                        EndDialog(hDlg, FALSE);
                        return TRUE;
        case IDCANCEL:
            EndDialog(hDlg, FALSE);
            return FALSE;
```

Y de momento esto es todo lo que diremos sobre CheckBoxes.

## Ejemplo 14

# Capítulo 15 Control básico

## RadioButton

De nuevo estamos hablando de un estilo de botón.

Los RadioButtons sólo pueden tomar dos valores, encendido y apagado. Se usan típicamente para leer opciones que sólo tienen un número limitado y pequeño de posibilidades y sólo un valor posible, como por ejemplo: sexo (hombre/mujer), estado civil (soltero/casado/viudo/divorciado), etc.

Es necesario agrupar usando un [GroupBox](#), al menos dos, y con frecuencia tres o más controles de este tipo. No tiene sentido colocar un solo control RadioButton, ya que al menos uno de cada grupo debe estar activo. Tampoco es frecuente agrupar dos, ya que para eso se puede usar un único control CheckBox. Tampoco se agrupan demasiados, ya que ocupan mucho espacio, en esos casos es mejor usar un ComboBox o un ListBox.

El aspecto normal es el de un pequeño círculo con un texto a uno de los lados, normalmente a la derecha. Cuando está activo se muestra el círculo relleno, cuando no lo está, aparece vacío. También es posible mostrar el RadioButton como un botón corriente, en ese caso, al activarse se quedará pulsado.

## Ficheros de recursos

Vamos a mostrar algunos de los posibles aspectos de los RadioButtons:

```
#include <windows.h>
#include "win015.h"
```

```

Menu MENU
BEGIN
    POPUP "&Principal"
    BEGIN
        MENUITEM "&Diálogo", CM_DIALOGO
    END
END

DialogoPrueba DIALOG 0, 0, 179, 89
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
CAPTION "RadioButtons"
FONT 8, "Helv"
BEGIN
    CONTROL "Grupo 1", ID_GRUPO1, "BUTTON",
        BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP,
        4, 5, 76, 52
    CONTROL "RadioButton 1", ID_RADIOBUTTON1, "BUTTON",
        BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        11, 15, 60, 12
    CONTROL "RadioButton 2", ID_RADIOBUTTON2, "BUTTON",
        BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        11, 28, 60, 12
    CONTROL "RadioButton 3", ID_RADIOBUTTON3, "BUTTON",
        BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        11, 41, 60, 12
    CONTROL "Grupo 2", ID_GRUPO2, "BUTTON",
        BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP,
        89, 5, 76, 52
    CONTROL "RadioButton 4", ID_RADIOBUTTON4, "BUTTON",
        BS_RADIOBUTTON | BS_PUSHLIKE | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        96, 15, 60, 12
    CONTROL "RadioButton 5", ID_RADIOBUTTON5, "BUTTON",
        BS_RADIOBUTTON | BS_PUSHLIKE | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        96, 28, 60, 12
    CONTROL "RadioButton 6", ID_RADIOBUTTON6, "BUTTON",
        BS_RADIOBUTTON | BS_PUSHLIKE | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        96, 41, 60, 12
    CONTROL "Aceptar", IDOK, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        8, 69, 50, 14
    CONTROL "Cancelar", IDCANCEL, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,

```

```
68, 69, 50, 14  
END
```

Para ver más detalles acerca de este tipo de controles ver [controles button](#).

Como se puede observar, un RadioButton es un tipo de botón con uno de los siguientes estilos **BS\_RADIOBUTTON** o **BS\_AUTORADIOBUTTON**.

También se puede ver que hemos agrupado los controles RadioButton en dos grupos de tres botones. Cada grupo empieza con un control GroupBox con el estilo **WS\_GROUP**. Más adelante veremos cómo trabajan en conjunto los GroupBoxes y los RadioButtons.

```
CONTROL "Normal", ID_RADIOBUTTON1, "BUTTON", BS_RADIOBUTTON  
| WS_CHILD | WS_VISIBLE | WS_TABSTOP, 12, 4, 60, 12
```

- **CONTROL** es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los RadioButtons será el texto que aparecerá acompañando al círculo o en el interior del botón y que servirá para identificar el valor a elegir.
- id es el identificador del control. El identificador será necesario en algunos casos para procesar los comandos procedentes del RadioButton, en el caso de los no automáticos será la única forma de tratarlos.
- class es la clase de control, en nuestro caso "BUTTON".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un [estilo button](#) y varios [de ventana](#):
  - **BS\_RADIOBUTTON**: Indica que se trata de un RadioButton no automático.
  - **BS\_AUTORADIOBUTTON**: Indica que se trata de un RadioButton automático.

- **BS\_PUSHLIKE**: Indica que se trata de un RadioButton con la apariencia de un botón corriente.
- **BS\_LEFTTEXT**: Indica que el texto del RadioButton se sitúa a la izquierda de la caja.
- **BS\_FLAT**: Indica apariencia plana, sin las sombras que simulan 3D.
- **WS\_CHILD**: crea el control como una ventana hija.
- **WS\_VISIBLE**: crea una ventana inicialmente visible.
- **WS\_TABSTOP**: para que cuando el foco cambie de control al pulsar TAB, pase por este control.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

## Iniciar controles RadioButton

Los controles RadioButton necesitan ser inicializados. Para este ejemplo también usaremos variables globales para almacenar los valores de las variables que se pueden editar con los RadioButtons. En general se necesita una única variable para cada grupo de RadioButtons.

```
// Datos de la aplicación
typedef struct stDatos {
    int Grupo1;
    int Grupo2;
} DATOS;
```

Crearemos una estructura de datos estática en nuestro procedimiento de ventana, y le daremos valores iniciales al procesar el mensaje **WM\_CREATE**:

```
static DATOS Datos;
...
```

```

case WM_CREATE:
    hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
    /* Inicialización */
    Datos.Grupo1 = 1;
    Datos.Grupo2 = 2;
    return 0;

```

Finalmente, usaremos la función [DialogBoxParam](#) para crear el diálogo, y pasaremos en el parámetro lParam un puntero a la estructura de datos.

```

DialogBoxParam(hInstance, "DialogoPrueba",
    hwnd, DlgProc, (LPARAM)&Datos);

```

Como siempre, para establecer los valores iniciales de los controles CheckBox usaremos el mensaje [WM\\_INITDIALOG](#) del procedimiento de diálogo.

Para eso usaremos la función [CheckRadioButton](#) o el mensaje [BM\\_SETCHECK](#), en este último caso, emplearemos la función [SendDlgItemMessage](#). Usar el mensaje implica enviar un mensaje al menos a dos controles RadioButton del grupo, el que se activa y el que se desactiva.

De nuevo ilustraremos el ejemplo usando los dos métodos:

```

static DATOS *Datos;
...
case WM_INITDIALOG:
    Datos = (DATOS*)lParam;
    // Estado inicial de los radiobuttons
    CheckRadioButton(hDlg, ID_RADIOBUTTON1,
ID_RADIOBUTTON3,
        ID_RADIOBUTTON1+Datos->Grupo1-1);
    // Usando mensajes:
    SendDlgItemMessage(hDlg, ID_RADIOBUTTON4,
        BM_SETCHECK, (WPARAM)BST_UNCHECKED, 0);
    SendDlgItemMessage(hDlg, ID_RADIOBUTTON5,
        BM_SETCHECK, (WPARAM)BST_UNCHECKED, 0);
    SendDlgItemMessage(hDlg, ID_RADIOBUTTON6,

```



```
        BM_SETCHECK, (WPARAM)BST_UNCHECKED, 0);  
    SendDlgItemMessage(hDlg, ID_RADIOBUTTON4+Datos->Grupo2-1,  
        BM_SETCHECK, (WPARAM)BST_CHECKED, 0);  
    SetFocus(GetDlgItem(hDlg, ID_RADIOBUTTON1));  
    return FALSE;
```

Es muy importante asignar identificadores correlativos a los controles de cada grupo. Esto nos permite por una parte usar la función [CheckRadioButton](#), tanto como expresiones como `ID_RADIOBUTTON1+Datos>Grupo1-1`. (Si hubiéramos empezado por cero para el primer control del grupo, no sería necesario restar uno).

Usando los mensajes nos vemos obligados a quitar la marca a todos los controles del grupo 2. Esto es porque desconocemos el estado inicial de los controles. En este caso es mucho mejor usar la función que el mensaje.

Con esto, el estado inicial de los RadioButtons será correcto.

## Procesar mensajes de los RadioButtons

En ciertos casos, será necesario procesar algunos de los mensajes procedentes de los RadioButtons.

Concretamente, en el caso de los RadioButtons no automáticos, su estado no cambia cuando el usuario actúa sobre ellos, sino que será el programa quien deba actualizar ese estado.

Para actualizar el estado de los RadioButtons cada vez que el usuario actúe sobre ellos debemos procesar los mensajes [WM\\_COMMAND](#) procedentes de ellos.

Como sucedía con los CheckBoxes, la primera intención puede que sea modificar las variables almacenadas en la estructura Datos para adaptarlas a los nuevos valores. Pero recuerda que es posible que el usuario pulse el botón de "Cancelar". En ese caso, los valores de la estructura Datos no deberían cambiar.

De nuevo tenemos dos opciones. Una es usar variables auxiliares para almacenar el estado actual de los RadioButtons. Otra es leer el estado de los controles directamente. Este último sistema es más seguro, ya que previene el que las variables auxiliares y los controles tengan valores diferentes.

Para leer el estado de los controles tenemos dos posibilidades, como siempre: usar la función [IsDlgButtonChecked](#) o el mensaje [BM\\_GETCHECK](#). Veremos las dos opciones.

Con funciones:

```
switch(LOWORD(wParam)) {
    case ID_RADIOBUTTON4:
    case ID_RADIOBUTTON5:
    case ID_RADIOBUTTON6:
        CheckRadioButton(hDlg, ID_RADIOBUTTON4,
ID_RADIOBUTTON6,
            LOWORD(wParam));
        return TRUE;
```

Con mensajes:

```
switch(LOWORD(wParam)) {
    case ID_RADIOBUTTON4:
    case ID_RADIOBUTTON5:
    case ID_RADIOBUTTON6:
        SendDlgItemMessage(hDlg, ID_RADIOBUTTON4,
BM_SETCHECK,
            (WPARAM)BST_UNCHECKED, 0);
        SendDlgItemMessage(hDlg, ID_RADIOBUTTON5,
BM_SETCHECK,
            (WPARAM)BST_UNCHECKED, 0);
        SendDlgItemMessage(hDlg, ID_RADIOBUTTON6,
BM_SETCHECK,
            (WPARAM)BST_UNCHECKED, 0);
        SendDlgItemMessage(hDlg, LOWORD(wParam),
BM_SETCHECK,
            (WPARAM)BST_CHECKED, 0);
        return TRUE;
```



```
Datos->Grupo2 = 2;  
    else  
Datos->Grupo2 = 3;  
    EndDialog(hDlg, FALSE);  
    return TRUE;  
case IDCANCEL:  
    EndDialog(hDlg, FALSE);  
    return FALSE;
```

Con esto queda cerrado de momento el tema de los RadioButtons.

## Ejemplo 15

# Capítulo 16 El GDI

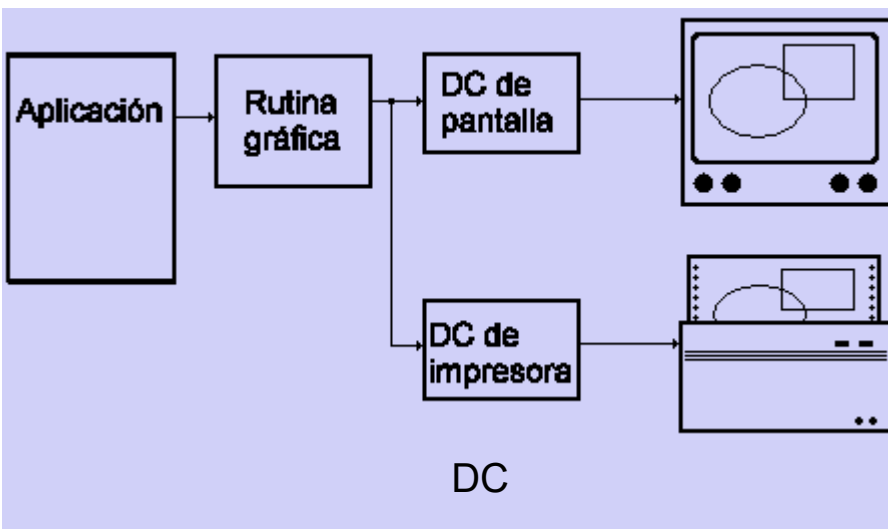
Ahora ya dominamos los controles básicos, así que pasaremos a un capítulo muy amplio, pero mucho más interesante.

El GDI (Graphics Device Interface), es el interfaz de dispositivo gráfico, que contiene todas las funciones y estructuras necesarias que nos permiten comunicar nuestras aplicaciones con cualquier dispositivo gráfico de salida conectado a nuestro ordenador: pantalla, impresora, plotter, etc.

Veremos que podremos trazar líneas, curvas, figuras cerradas, polígonos y mapas de bits. Además, podremos controlar características como colores, aspectos de líneas y tramas de superficies rellenas, mediante objetos como plumas, pinceles y fuentes de caracteres.

Las aplicaciones Windows dirigen las salidas gráficas a lo que se conoce como un Contexto de Dispositivo, abreviado como DC. Cada DC se crea para un dispositivo concreto.

Un DC es una de las estructuras del GDI, que contiene información sobre el dispositivo, como las opciones seleccionadas, o modos de funcionamiento.



La aplicación crea un DC mediante una función del GDI, y éste devuelve un manipulador de DC (hDC), que se usará en las siguientes

llamadas para identificar el dispositivo que recibirá la salida. Mediante el DC, la aplicación también puede obtener ciertas capacidades del dispositivo, como las dimensiones del área de impresión, la resolución, etc.

La salida puede enviarse directamente al DC del dispositivo físico, o bien a un dispositivo lógico, que se crea en memoria o en un metafile. La ventaja de usar un dispositivo lógico es que se almacena la salida completa y posteriormente puede enviarse al cualquier dispositivo físico.

También existen funciones para seleccionar diferentes modos y opciones del dispositivo. Eso incluye colores del texto y fondo, plumas y pinceles de diferentes colores y texturas para trazar líneas o rellenar superficies, y lo que se conoce como operaciones binarias de patrones (Binary Raster Operations), que indican cómo se combinan las nuevas salidas gráficas con las existentes previamente. También existen varios sistemas de mapeo de coordenadas, para traducir las coordenadas usadas en la aplicación con las el sistema de coordenadas del dispositivo.

## **Objetos del GDI**

Windows usa objetos y manipuladores para acceder a los recursos del sistema, en lo que se refiere al GDI existen los siguientes objetos:

Mapas de bits, pinceles, fuentes, plumas, plumas extendidas, regiones, contextos de dispositivos, contextos de dispositivo de memoria, metafiles, contextos de dispositivo de metafiles, metafiles mejorados y contextos de dispositivo de metafiles mejorados.

Veremos cada uno de estos objetos, algunos en los próximos capítulos, los que denominaremos básicos: mapas de bits, pinceles, fuentes, plumas y dispositivos de contexto. El resto en capítulos más avanzados.

El API proporciona funciones para crear objetos y manipuladores de objetos, cerrar un manipulador de objeto y destruir un objeto.

Los objetos y los manipuladores consumen memoria, es muy importante cerrar los manipuladores y destruir los objetos cuando no se necesitan más, no hacerlo puede ralentizar el sistema e incluso volverlo inestable.

# Capítulo 17 Objetos básicos del GDI: El Contexto de dispositivo, DC

La estructura del DC, como la del resto de los objetos del GDI, no es accesible directamente desde las aplicaciones. Contiene los valores de ciertos atributos que determinan la posición y apariencia de la salida hacia un dispositivo.

Cada vez que se crea un DC, cada atributo tiene un valor por defecto conocido. Los atributos almacenados en el DC, y sus valores por defecto son:

Atributo	Valor por defecto
Color de fondo	Blanco (Depende del seleccionado en el panel de control)
Modo del fondo	Opaco
Mapa de bits (bitmap)	Ninguno
Pincel (Brush)	WHITE_BRUSH (Blanco)
Origen para el pincel	(0,0)
Región de recorte (Clipping region)	Toda la superficie visible de la ventana
Paleta de colores	DEFAULT_PALETTE (paleta por defecto)
Posición actual de la pluma	(0,0)
Origen del dispositivo	Esquina superior derecha de la ventana o del área de cliente.
Modo de trazado	R2_COPYPEN (sustitución del valor actual en pantalla)
Fuente	SYSTEM_FONT (fuente del sistema)



Espacio entre caracteres	0
Modo de mapeo	MM_TEXT (coordenadas de texto)
Pluma	BLACK_PEN (negra)
Modo para el relleno de polígonos	ALTERNATE (alternativo)
Modo de estiramiento (Stretching mode)	BLACKONWHITE
Color de texto	Negro
Tamaño del Viewport	(1,1)
Origen del Viewport	(0,0)
Tamaño de la ventana	(1,1)
Origen de la ventana	(0,0)

Existen funciones para acceder a los valores del DC, mediante un manipulador, ya sea para leer sus valores o para modificarlos. En próximos capítulos veremos algunas de ellas.

## Actualizar el área de cliente de una ventana, el mensaje WM\_PAINT

Todos los ejemplos estarán centrados en el proceso del mensaje `WM_PAINT`, que le indica al procedimiento de ventana que parte o toda su superficie debe ser actualizada.

En nuestros primeros ejemplos y mientras veamos los diferentes objetos, funciones y estructuras, no necesitaremos crear un DC, simplemente obtendremos un manipulador para el DC de la ventana de nuestro programa. Cada ventana tiene asociados varios DCs. A nosotros nos interesa el que está asociado al área de cliente.

Para actualizar el área de cliente de nuestra ventana tendremos que seguir una especie de "ritual". Eso es imprescindible, y a menudo se siguen estas "fórmulas" en muchos procesos de los programas Windows, muchas veces sin saber el motivo de cada paso. Pero en este curso intentaremos explicar el por qué de cada uno de los pasos, y que acciones desempeña cada uno de ellos.

Lo que sigue es un ejemplo muy sencillo de cómo se procesa un mensaje [WM\\_PAINT](#):

```
HDC hDC;  
PAINTSTRUCT ps;  
...  
    case WM_PAINT:  
        hDC = BeginPaint(hwnd, &ps);  
        Ellipse(hDC, 10, 10, 50, 50);  
        EndPaint(hwnd, &ps);  
        break;
```

Hay más formas de enviar salidas a una ventana, y las veremos más adelante, pero la más sencilla es ésta.

Utilizaremos la función [BeginPaint](#) para poder obtener un DC a nuestra ventana. Esta función además ajusta el área de "clipping", eso sirve para que sólo se actualice la parte de la ventana que sea necesario. Por ejemplo, imaginemos que nuestra ventana está parcialmente oculta por otras, y que la pasamos a primer plano. En ese caso, el área de "clipping" será sólo la correspondiente a las partes que estaban ocultas previamente, el resto de la ventana no se actualizará. Esto ahorra bastante tiempo de proceso, ya que dibujar gráficos suele ser un proceso lento.

El segundo parámetro es un puntero a una estructura [PAINTSTRUCT](#) que es necesario para la llamada. Los datos obtenidos mediante esa estructura pueden sernos útiles en algunas ocasiones, pero de momento sólo la usaremos para obtener un DC del área de cliente de la ventana.

Una vez que hemos obtenido un DC para nuestra ventana, podremos utilizarlo para dibujar en ella. En este ejemplo sólo se dibuja una circunferencia, usando al función [Ellipse](#). En próximos ejemplos iremos complicando más esa parte.

Por último, una vez que hemos terminado de pintar, liberamos el DC usando la función [EndPaint](#). Es muy importante hacer esto cada vez que obtengamos un DC mediante [BeginPaint](#).

# Colores

Windows almacena los colores en un entero de 32 bits en formato RGB, hay un tipo definido para eso, se llama **COLORREF**. Los 8 bits de menor peso se usan para almacenar la componente de rojo, los 8 siguientes para el verde, los 8 siguientes para el azul y el resto no se usa, y debe ser cero.

Para referenciar o asignar un color se usa la macro **RGB**, que admite tres parámetros de tipo BYTE, cada uno de ellos indica la intensidad relativa de una de las componentes: rojo, verde o azul, y devuelve un **COLORREF**.

También es posible obtener cada una de las componentes de un **COLORREF** mediante las macros **GetRValue**, **GetGValue** y **GetBValue**.

# Capítulo 18 Objetos básicos del GDI: La pluma (Pen)

La pluma se utiliza para trazar líneas y curvas. En este capítulo veremos cómo crearlas, seleccionadas y destruirlas, y cómo elegir el estilo, grosor y color para una pluma.

El proceso con los objetos es siempre el mismo, hay que crearlos, seleccionarlos y, cuando ya no se necesiten, destruirlos.

## Plumas de Stock

La única excepción a lo dicho antes es un pequeño almacén de objetos que usa el sistema y que están disponibles para usar en nuestros programas.

El ejemplo anterior funciona gracias a ello, ya que usa la pluma por defecto del DC. En el caso de los objetos de stock, no es necesario crearlos ni destruirlos (aunque esto último no está prohibido), siempre podemos obtener un manipulador y seleccionarlo para usarlo.

Concretamente, en el caso de las plumas, existen tres en el stock:

Valor	Significado
BLACK_PEN	Pluma negra
NULL_PEN	Pluma nula
WHITE_PEN	Pluma blanca

Para obtener un manipulador para una de esos objetos de stock se usa la función [GetStockObject](#). Dependiendo del valor que usemos obtendremos un tipo de objeto diferente.

# Plumas cosméticas y geométricas

Existen dos categorías de plumas.

Las cosméticas se crean con un grosor expresado en unidades de dispositivo, es decir, las líneas que se tracen con estas plumas tendrán siempre el mismo grosor. Estas plumas sólo tienen tres atributos: grosor, color y estilo. Las plumas de stock son de este tipo.

Las geométricas se crean con un grosor expresado en unidades lógicas. Esto significa que el grosor de las líneas puede ser escalado, y depende de las transformaciones actuales del "mundo actual" (veremos esto en siguientes capítulos). Además de los tres atributos que también tienen las plumas cosméticas, las geométricas poseen otros cuatro: plantilla, trama opcional, estilos para extremos y para uniones.

Las líneas trazadas con plumas cosméticas son entre tres y diez veces más rápidas que las de las geométricas.

## Crear una pluma

Si no vamos a utilizar una pluma de stock, (la verdad es que están muy limitadas), podemos crear nuestras propias plumas.

Para ello disponemos de la función [CreatePen](#). Esta función también nos devuelve un manipulador, en este caso de pluma, que podremos usar posteriormente para seleccionar la nueva pluma.

[CreatePen](#) nos pide tres parámetros. El primero es el estilo de línea, podemos elegir uno de los siguientes valores:

Estilo	Descripción
<a href="#">PS_SOLID</a>	Las líneas serán continuas y sólidas.
<a href="#">PS_DASH</a>	Líneas de trazos. Este estilo sólo es válido cuando el ancho de la pluma sea uno o menos en unidades de dispositivo.
<a href="#">PS_DOT</a>	Líneas de puntos. Este estilo sólo es válido cuando el ancho de la pluma sea uno o menos

en unidades de dispositivo.

#### PS\_DASHDOT

Líneas alternan puntos y trazos. Este estilo sólo es válido cuando el ancho de la pluma sea uno o menos en unidades de dispositivo.

#### PS\_DASHDOTDOT

Líneas alternan líneas y dobles puntos. Este estilo sólo es válido cuando el ancho de la pluma sea uno o menos en unidades de dispositivo.

#### PS\_NULL

Las líneas son invisibles.

#### PS\_INSIDEFRAME

Las líneas serán sólidas. Cuando ésta pluma se usa en cualquier función de dibujo del GDI que requiera un rectángulo que sirva como límite, las dimensiones de la figura se reducirán para que se ajusten por completo al interior del rectángulo, teniendo en cuenta el grosor de la pluma. Esto sólo se aplica a plumas geométricas.

El segundo parámetro es el grosor de la línea en unidades lógicas, si se especifica un grosor de cero, la línea tendrá un pixel de ancho.

El último parámetro es un [COLORREF](#), que será el color de las líneas.

Evidentemente, [CreatePen](#) sólo puede crear plumas cosméticas.

Existen otras dos funciones para crear plumas: [CreatePenIndirect](#) y [ExtCreatePen](#).

La primera sirve también para crear plumas cosméticas, pero lo hace a través de una estructura [LOGPEN](#), que almacena en su interior los mismos parámetros que se pasan a la función [CreatePen](#). Esto puede ser útil cuando creemos muchas plumas distintas o varias con muy pocas diferencias.

La segunda nos permite crear tanto plumas cosméticas como geométricas. De momento no veremos este tipo de plumas, las dejaremos para capítulos más avanzados.

## Seleccionar una pluma

Aunque podemos disponer de un repertorio de manipuladores de pluma, sólo puede haber una activa en cada momento, para seleccionar la pluma activa se usa la función [SelectObject](#).

En realidad, ésta función sirve para seleccionar cualquier tipo de objeto, no sólo plumas. El tipo de objeto seleccionado depende el parámetro que se pase a la función. El nuevo objeto seleccionado reemplaza al actual, y se devuelve el manipulador del objeto seleccionado anteriormente.

Se debe guardar el manipulador de la pluma por defecto seleccionada antes de cambiarla por primera vez, y restablecerlo antes de terminar el procedimiento de pintar.

## Destruir una pluma

Por último, cuando ya no necesitemos más los manipuladores de plumas, debemos destruirlos, con el fin de liberar la memoria usada para almacenarlos. Esto se hace mediante la función [DeleteObject](#).

## Ejemplo 16

# Capítulo 19 Funciones para el trazado de líneas

El GDI dispone de un repertorio de funciones para el trazado de líneas bastante completo

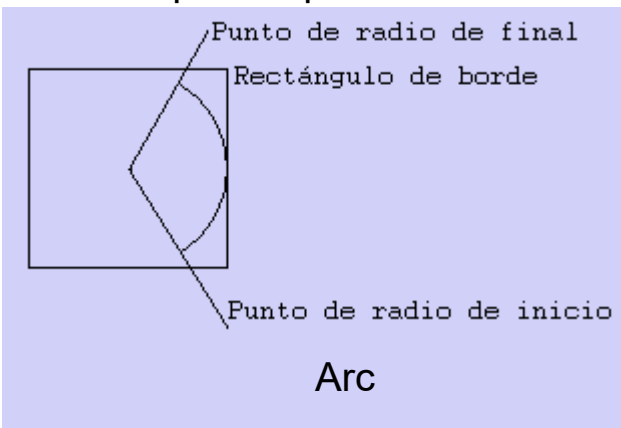
Función	Tipo de línea
MoveToEx	Actualiza la posición actual del cursor gráfico, opcionalmente obtiene la posición anterior.
LineTo	Traza una línea desde la posición actual del cursor al punto indicado.
ArcTo	Traza un arco de elipse.
PolylineTo	Traza uno o más trazos de líneas rectas.
PolyBezierTo	Traza una o más curvas Bézier.
AngleArc	Traza un segmento de arco de circunferencia.
Arc	Traza un arco de elipse.
Polyline	Traza una serie de segmentos de recta que conectan los puntos de un array.
PolyBezier	Traza una o más curvas Bézier.
GetArcDirection	Devuelve la dirección de arco del DC actual.
SetArcDirection	Cambia la dirección del ángulo para el trazado de arcos y rectángulos.
LineDDA	Traza una línea, pero permite a una función de usuario decidir qué pixels se mostrarán.
LineDDAProc	Función callback de la aplicación que procesa las coordenadas recibidas de LineDDA.
PolyDraw	Traza una o varias series de líneas y curvas Bézier.
PolyPolyLine	Traza una o varias series de segmentos de recta conectados.



La mayoría de éstas funciones no requieren mayor explicación, sobre todo las que trazan arcos y líneas rectas.

## Trazado de arcos, función Arc

Para trazar un arco mediante la función Arc, se parte de una circunferencia inscrita en el rectángulo de borde. Los puntos de inicio y final del arco se obtienen de los cortes de los radios definidos por los puntos de radio de inicio y final:



La función ArcTo es idéntica, salvo que se traza una línea desde la posición actual del cursor gráfico hasta el punto de inicio del arco.

## Curvas Bézier

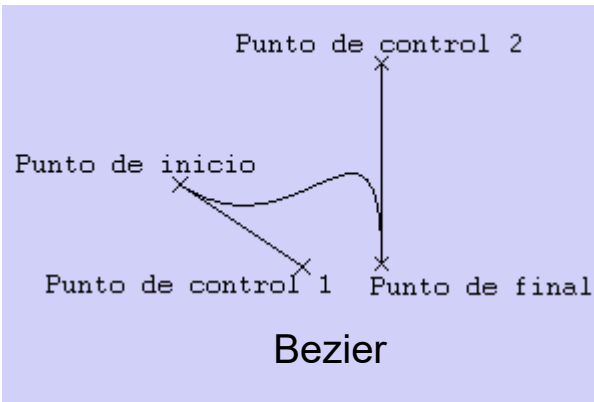
Las curvas Bézier son una forma de definir curvas irregulares mediante métodos matemáticos. Para definir una curva Bézier sólo se necesitan cuatro puntos: los puntos de inicio y final, y dos puntos de control.

En el gráfico se representa una curva Bézier y las líneas auxiliares, que sólo se muestran como ayuda. El punto de inicio y el punto de control 1 refinan una recta que es tangente a la curva en el punto de inicio. La curva tiende a aproximarse al punto de control 1.

Análogamente, el punto de final y el punto de control 2 refinan otra recta, que es tangente a la curva en el punto de final. La curva también tiende a acercarse al punto de control 2.

Las ecuaciones que definen la curva no son demasiado complejas, pero escapan al objetivo de este curso, se estudiarán las curvas de Bézier con detalle en un artículo separado.

## Funciones Poly<tipo>



Todas las funciones con el prefijo Poly trabajan de un modo parecido. Se basan en un array de puntos para definir un conjunto de líneas que se trazan una a continuación de otra.

Polyline o PolylineTo trazan un conjunto de segmentos

rectos, PolyBezier y PolyBezierTo, un conjunto de curvas Bézier. PolyDraw, varios conjuntos de líneas y curvas Bézier, y PolyPolyline, varios conjuntos de líneas rectas.

## Función LineDDA y funciones callback LineDDAProc

[LineDDA](#) nos permite personalizar el trazado de segmentos de líneas rectas. Mediante la definición de funciones propias del tipo LineDDAProc, podemos procesar cada punto de la línea y decidir cómo visualizarlo. Podemos cambiar el color, ignorar ciertos puntos, y en general, aplicar la modificación que queramos. Esto nos permite trazar líneas de varios colores o con distintas tramas, etc.

Veamos un ejemplo sencillo, definiremos una función LineDDAProc para trazar líneas que alternen 10 pixels rojos y 10 verdes.

```
struct DatosDDA1 {
    int cuenta;
    HDC hdc;
};

VOID CALLBACK FuncionDDA1(int X, int Y, LPARAM datos)
{
    struct DatosDDA1 *dato = (struct DatosDDA1 *)datos;

    // Función que pinta líneas con 10 pixels alternados
    rojos y verdes
```

```

dato->cuenta++;
if(dato->cuenta >= 20) dato->cuenta = 0; // Mantenemos
cuenta entre 0 y 19
if(dato->cuenta < 10)
    SetPixel(dato->hdc, X, Y, RGB(0,255,0));
else
    SetPixel(dato->hdc, X, Y, RGB(255,0,0));
}

```

Hemos definido una función callback para que muestre cada punto en función de los datos almacenados en una estructura DatosDDA1, diseñada por nosotros. En ella almacenamos un valor "cuenta" que nos ayuda a decidir el color de cada pixel, y un manipulador de DC para poder activar pixels, usando [SetPixel](#), en la ventana:

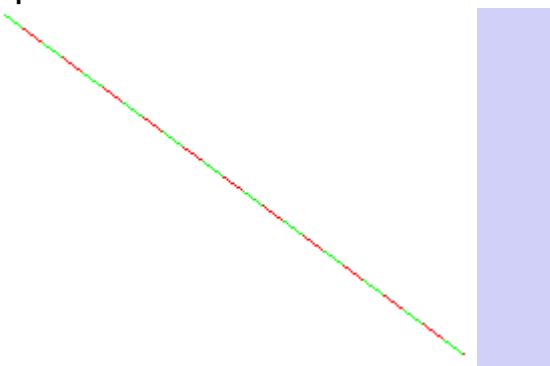
```

...
struct DatosDDA1 datos = {0, hdc};

LineDDA(10,10, 240,180, FuncionDDA1, (LPARAM)&datos);
...

```

Ahora podemos llamar a la función [LineDDA](#), indicando los puntos de inicio y final de la línea, la función que usaremos para decidir el color de cada punto, y un puntero a la estructura de datos que esa función necesita.



Línea trazada con LineDDA

## Ejemplo 17

Basándonos en el último programa de ejemplo, vamos a hacer otro que muestre cada una de éstas funciones y cómo se usan.

Añadiremos un menú para poder ver cada función por separado.

**Nota:**

Algunas de las funciones del API usadas en el ejemplo no están disponibles en Windows 95 y Win32s, por ejemplo: [PolyDraw](#), [ArcTo](#) y [AngleArc](#). Esto no impide que el programa se pueda compilar y ejecutar, pero algunas funciones no funcionarán.

# Capítulo 20 Objetos básicos del GDI: El pincel (Brush)

El pincel se utiliza para rellenar superficies y figuras cerradas. En este capítulo veremos cómo crearlos, seleccionarlos y destruirlos, y cómo elegir el estilo, textura y color para un pincel.

Al igual que vimos con las plumas, el proceso con todos los objetos es siempre el mismo, hay que crearlos, seleccionarlos y, cuando ya no se necesiten, destruirlos.

## Pinceles lógicos

Existen cuatro tipos distintos de pinceles lógicos.

- Sólidos, consisten en un único color continuo.
- Stock, pinceles predefinidos por el sistema.
- Hatch, tramas de líneas.
- Patrones, consisten en mapas de bits.

## Pinceles sólidos

Son los más simples, se usan para rellenar superficies con un único color uniforme. Para crear uno de estos pinceles se usa la función [CreateSolidBrush](#), que sólo requiere que se especifique el color del pincel.

## Pinceles de Stock

Del mismo modo que sucede con las plumas, también disponemos de un juego de pinceles de stock que podremos usar en nuestros programas.

En el caso de los objetos de stock, no es necesario crearlos ni destruirlos, siempre podemos obtener un manipulador y seleccionarlo para usarlo.

En el caso de los pinceles, existen siete en el stock:

Valor	Significado
BLACK_BRUSH	Pincel negra
DKGRAY_BRUSH	Pincel gris oscuro
GRAY_BRUSH	Pincel gris
HOLLOW_BRUSH	Pincel hueco
LTGRAY_BRUSH	Pincel gris claro
NULL_BRUSH	Pincel nulo (equivale a HOLLOW_BRUSH)
WHITE_BRUSH	Pincel blanco

Ya vimos que para obtener un manipulador para una de esos objetos de stock se usa la función [GetStockObject](#). Dependiendo del valor que usemos obtendremos un tipo de objeto diferente.

## Pinceles de tramas (Hatch)

Consisten en tramas de líneas paralelas, que permiten crear superficies ralladas.

Para crear pinceles tramados se usa la función [CreateHatchBrush](#). Además del color podemos escoger entre varias tramas distintas.

Existen seis tipos de tramas predefinidas, accesibles mediante constantes:

Valor	Significado
HS_BDIAGONAL	Trama de líneas diagonales a 45° descendentes de izquierda a derecha.
HS_CROSS	Trama de líneas horizontales y verticales.
HS_DIAGCROSS	Trama de líneas diagonales a 45° cruzadas.

**HS\_FDIAGONAL** Trama de líneas diagonales a 45°  
ascendentes de izquierda a  
derecha.

**HS\_HORIZONTAL** Trama de líneas horizontales.

**HS\_VERTICAL** Trama de líneas verticales.

## Pinceles de patrones

Este tipo de pincel se crea a partir de un mapa de bits, así que en realidad puede ser cualquier tipo de imagen. Antes de poder crear uno de estos pinceles tendremos que disponer de un mapa de bits, que, como comentamos antes, es otro de los objetos de GDI que podemos manejar.

Supondremos que ya sabemos crear un objeto de mapa de bits, aunque explicaremos cómo hacerlo en próximos capítulos.

Para crear un pincel de patrón disponemos de tres funciones: **CreatePatternBrush**, **CreateDIBPatternBrushPt** y **CreateDIBPatternBrush**. La primera sólo requiere un manipulador de un mapa de bits. La segunda y la tercera permiten usar mapas de bits independientes del dispositivo, lo cual proporciona mayor control sobre la paleta de colores.

Para poder usar este tipo de pinceles hay que estar algo más familiarizado con los mapas de bits, volveremos sobre este tema cuando hayamos visto el objeto bitmap.

## Crear un pincel

Además de las funciones mencionadas para crear pinceles de los distintos tipos lógicos mencionados, existe una función más para crear pinceles: **CreateBrushIndirect**.

Esta función sirve para crear pinceles lógicos, pero lo hace a través de una estructura **LOGBRUSH**, que almacena en su interior los parámetros necesarios para crear un pincel sólido, de trama o de patrón.

## Seleccionar un pincel

Con los pinceles sucede lo mismo que con las plumas, aunque podemos tener de un repertorio de manipuladores de pincel, sólo puede haber uno activo en cada momento, para seleccionar el pincel activo se usa la función [SelectObject](#).

Como ya hemos dicho el tipo de objeto seleccionado depende del parámetro que se pase a la función. El nuevo objeto seleccionado reemplaza al actual, y se devuelve el manipulador del objeto seleccionado anteriormente.

Se debe guardar el manipulador del pincel por defecto seleccionado antes de cambiarlo por primera vez, y restablecerlo antes de terminar el procedimiento de pintar.

## Destruir un pincel

Por último, cuando ya no necesitemos más los manipuladores de pinceles, debemos destruirlos, con el fin de liberar la memoria usada para almacenarlos. Esto se hace mediante la función [DeleteObject](#).

## Ejemplo 18



# Capítulo 21 Funciones para el trazado de figuras rellenas

Veremos ahora el repertorio de funciones para el trazado de figuras cerradas rellenas. Para trazar estas figuras se usa una pluma para el borde y un pincel para los interiores.

Función	Tipo de figura
Chord	Traza una figura definida por el corte de una elipse y una recta secante y la rellena.
Ellipse	Traza una elipse rellena.
FillRect	Rellena un rectángulo, sin trazar el borde.
FrameRect	Traza un borde alrededor de un rectángulo usando el pincel actual.
Pie	Traza un sector de elipse.
Polygon	Traza un polígono relleno.
PolyPolygon	Traza una serie de polígonos cerrados y rellenos.
Rectangle	Traza un rectángulo relleno.
RoundRect	Traza un rectángulo relleno con las esquinas redondeadas.

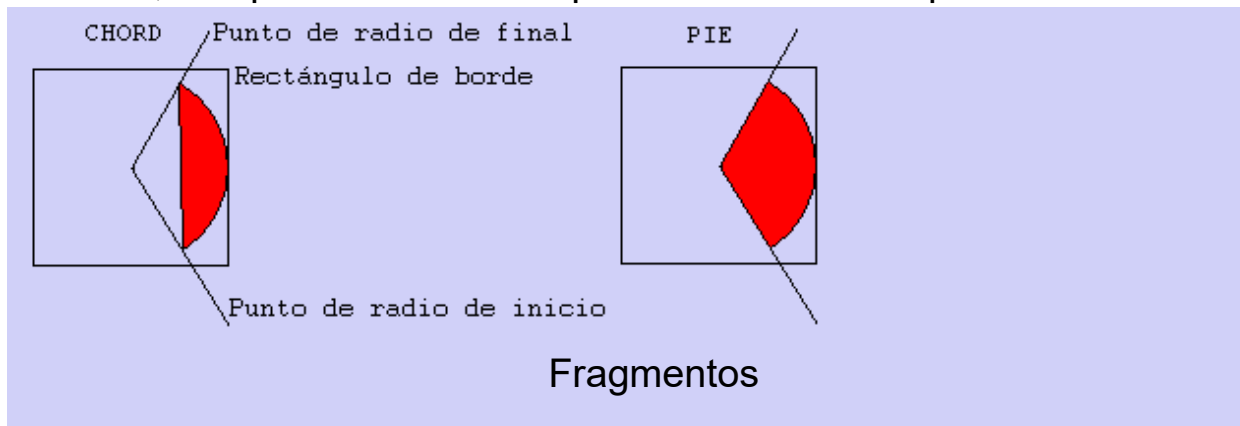
La mayoría de éstas funciones no requieren mayor explicación, los nombres y descripciones dan suficiente información sobre su cometido.

## Pintando trozos de elipses, funciones Chord y Pie

Existen dos funciones para trazar figuras rellenas partiendo de una elipse. Podemos clasificar estas figuras en función del número de trazos rectos que contienen.

Según ese criterio, si sólo hay un segmento recto se trata de la figura que se puede trazar con la función Chord, si tiene dos segmentos rectos se trata de la figura que se puede trazar con la función Pie.

Por ejemplo, las formas que se obtienen al partir una galleta son "chords", las que se obtienen al partir una tarta son "pies":



El "centro" de la elipse es el punto medio entre los dos focos, en el caso de la circunferencia, que en realidad es una elipse "degenerada", en la que los dos focos coinciden en un punto, ese punto es el centro.

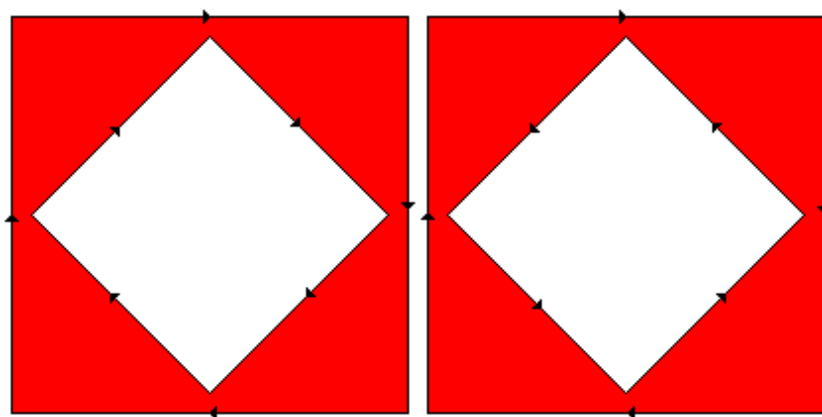
## Modos de relleno de polígonos

Existen dos modos de relleno de polígonos. En general la diferencia es mínima, y sólo se manifiesta en casos de polígonos complejos, con varias superposiciones.

Los dos modos de relleno son:

- Alternativo: rellena las áreas entre las líneas impares y pares de cada línea de rastreo. Para ver cómo funciona este modo, imaginemos que recorremos cada línea horizontal de la pantalla de izquierda a derecha. El espacio entre el borde y la primera línea del polígono se deja sin rellenar, el espacio entre la primera

línea y la segunda se rellena, el espacio entre la segunda y la tercera, si existe, se deja sin rellenar, etc. Por ejemplo:



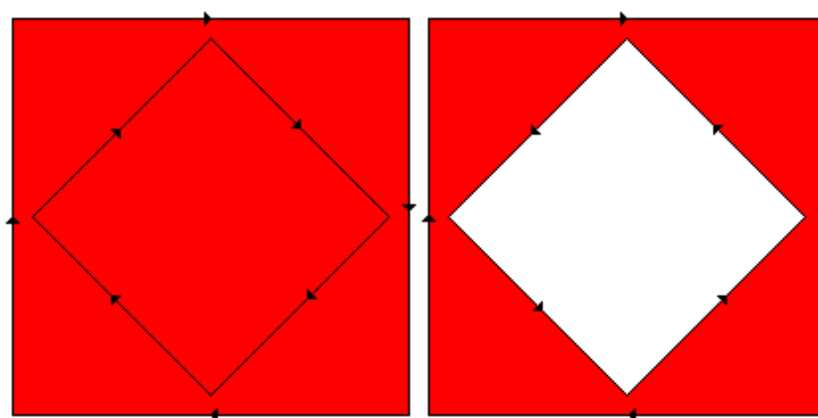
Alternado de relleno

- Tortuoso (winding): en este modo se asigna un número a cada región de la pantalla dependiendo del número de veces que se ha usado la pluma para trazar el polígono que la define. Hay que tener en cuenta la

dirección en que se recorre cada línea. Las regiones en que ese número no sea nulo, se rellenarán.

Por ejemplo, supongamos la figura siguiente:

Al seguir las líneas del cuadrado externo en el sentido de las flechas, cada uno de los dos cuadrados es rodeado una vez, a cada uno de ellos le asignamos un valor de winding igual a uno.



Relleno tortuoso

Al seguir las líneas de cuadrado interno, en el caso de la izquierda recorreremos el cuadrado interno en el mismo sentido que la primera vez, por lo tanto, le sumamos a esa figura una unidad a su valor de winding. En el caso de la figura de la derecha, la recorreremos en sentido contrario, por lo tanto le restamos una unidad al cuadrado

interno, es decir, que en el caso de la derecha, el valor winding del cuadrado interno es cero, y no se rellena.

Si seleccionamos el modo de llenado alternativo, el cuadrado interno no se rellenará nunca, como de hecho sucede en la primera imagen.

## **Ejemplo 19**

# Capítulo 22 Objetos básicos del GDI: La paleta (Palette)

El color es muy importante en Windows, y como todo, es un recurso que tiene sus limitaciones. Cada dispositivo tiene sus capacidades de colores, y el API proporciona funciones para conocer esas capacidades, así como manipularlos, elegirlos, activarlos, etc.

Tal vez, al menos a nivel de pantalla, ya no tenga mucho sentido un capítulo como el presente, las tarjetas gráficas actuales ya no tienen las limitaciones en cuanto a color que tenían hace unos años. Pero en el capítulo dedicado a los mapas de bits veremos que cuando se almacenan en disco o se transmiten, usar paletas nos ahorrará mucho espacio de almacenamiento y mucho tiempo en transmisiones.

## Capacidades de Color de los dispositivos

La capacidad de cada dispositivo: pantallas e impresoras, puede variar entre dos y miles o millones de colores. Eso suele ser debido a alguna propiedad física del dispositivo, por ejemplo, una impresora que sólo disponga de un cartucho de tinta negra, sólo podrá visualizar dos colores: blanco y negro. Del mismo modo, una tarjeta gráfica puede estar limitada por la memoria, y sólo disponer de 16 ó 256 colores, o un monitor en blanco y negro, que sólo disponga de una determinada gama de grises.

A menudo necesitaremos conocer las capacidades de los dispositivos con los que estamos trabajando, y así poder adaptar nuestras aplicaciones de modo que la apariencia de los resultados sea lo más parecido posible a lo que queremos.

Podemos averiguar el número de colores que disponibles en un dispositivo usando la función [GetDeviceCaps](#) con el parámetro [NUMCOLORS](#). El número obtenido será el número de colores disponibles para la aplicación.

## Definiciones de valores de color

Existen varios modos de codificar los valores de color, Windows usa la síntesis aditiva, expresando las intensidades relativas de los colores primarios: rojo, verde y azul. Para cada color se usan ocho bits, por lo tanto existen 256 valores posibles para cada componente, es decir un máximo de 16.777.216 colores. Los tres bytes forman un triplete RGB y se empaquetan en un entero de 32 bits, de los cuales, los 24 de menor peso se usan para las componentes y el resto a veces se usa para otras funciones que veremos más adelante.

Para almacenar valores de color se usa el tipo [COLORREF](#). Ya hemos usado este tipo antes, por ejemplo para crear plumas con [CreatePen](#) o para crear pinceles sólidos con [CreateSolidBrush](#), o en estructuras como [LOGPEN](#). Si se necesita extraer los valores individuales de los componentes de color de un valor [COLORREF](#) se pueden usar las macros [GetRValue](#), [GetGValue](#) y [GetBValue](#), para rojo, verde y azul respectivamente. También se pueden crear valores de color a partir de los componentes individuales usando la macro [RGB](#).

En las paletas lógicas, se usa la estructura [RGBQUAD](#) para definir valores de colores o para examinar valores de componentes.

## Aproximaciones de colores y mezclas de pixels (dithering)

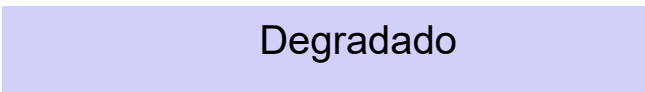
En cualquier caso, es posible usar colores sin preocuparse demasiado de las capacidades del dispositivo. Por ejemplo, nada

impide crear una pluma verde para una impresora en blanco y negro. Cuando pedimos un color que el dispositivo no admite, Windows escoge otro color entre los que sí puede generar, intentado elegir el más parecido. En nuestro ejemplo, Windows crearía una pluma negra.

El API dispone de la función [GetNearestColor](#) que admite como parámetro un valor de color y devuelve el más parecido que puede generar el dispositivo. Esto nos ayuda a predecir qué color obtendremos en cada caso.

Las aproximaciones siempre se usan cuando se eligen colores para plumas o para textos, pero cuando se eligen colores para pinceles sólidos Windows puede intentar simular el color mediante tramas de pixels de distintos colores elegidos entre los que el dispositivo sí puede generar.

En este ejemplo vemos cómo se simulan distintos tonos de verde mezclando los tonos de verde disponibles con negro:



No se dispone de ningún mecanismo para controlar cómo hace Windows estas

mezclas, ya que dependen del driver del dispositivo. Lo que sí podemos hacer es crear nuestros propios pinceles usando tramas de mapas de bits.

## Mezclas de colores (ROP)

Cuando dibujamos en pantalla usando una pluma o un pincel, no tenemos por qué limitarnos a activar pixels, es posible combinar el nuevo color con el color previo de cada pixel en pantalla. A esto se le denomina mezcla de colores.

Existen distintos modos de mezcla de primer plano, u operaciones binarias de rastreo, que determinan el modo en que se combinan los colores nuevos con los existentes en pantalla previamente. Es posible fundir colores, conservando todos los componentes de ambos colores; enmascarar colores, eliminando o

atenuando componentes no comunes; o enmascarar exclusivamente, eliminando o atenuando componentes comunes. Además hay variaciones sobre estas operaciones de mezcla básicas.

Veremos este tema en profundidad más adelante, en un capítulo dedicado a él.

Los colores obtenidos mediante la mezcla también se someten a aproximaciones de color. Si el color obtenido de una mezcla no puede ser mostrado por el dispositivo, Windows genera una aproximación. Como estas operaciones se hacen pixel a pixel, si el color en pantalla proviene de una mezcla de pixels, serán los pixels individuales los que se combinen.

Para seleccionar el modo de mezcla de primer plano se usa la función [SetROP2](#) y para recuperar la actual se usa [GetROP2](#).

**Nota:**

También existe un modo de mezcla de fondo, pero este modo no controla la mezcla de colores. En realidad especifica si el color de fondo será usado cuando se trazan líneas con estilos, pinceles de tramas y texto.

## Paletas de colores

Una paleta de colores es un conjunto que contiene valores de colores que pueden ser mostrados en el dispositivo de salida.

En Windows existen dos tipos de paletas: paletas lógicas y paletas de sistema. Dentro de las lógicas existe una especial, la paleta por defecto, que es la que se usa si el usuario no crea una.

Las paletas de colores se suelen usar en dispositivos que, aunque pueden generar muchos colores, sólo pueden mostrar o dibujar con un subconjunto de ellos en un momento dado. Para



estos dispositivos, Windows mantiene una paleta de sistema que permite almacenar y manejar los colores actuales del dispositivo.

Windows no permite acceder a la paleta de sistema directamente, en vez de eso, los accesos se hacen mediante una paleta lógica. Además, Windows crea una paleta por defecto para cada contexto de dispositivo. Como programadores, podemos usar los colores de la paleta por defecto o bien crear nuestra propia paleta lógica y asociarla al contexto de dispositivo.

Para determinar si un dispositivo soporta paletas de colores de puede comprobar el bit `RC_PALETTE` del valor `RASTERCAPS` devuelto por la función `GetDeviceCaps`.

## La paleta por defecto

Como ya hemos dicho, Windows asocia la paleta por defecto con un contexto cada vez que una aplicación crea un contexto para un dispositivo que soporte paletas de colores. De este modo Windows se asegura de que existen colores disponibles para usar en la aplicación sin necesidad de otras acciones.

La paleta por defecto normalmente tiene 20 entradas, pero ese número depende del dispositivo, y es igual al valor `NUMCOLORS` devuelto por `GetDeviceCaps`.

Los colores de la paleta por defecto dependen del dispositivo, en dispositivos de pantalla pueden ser los 16 colores estándar de VGA más cuatro definidos por Windows.

Cuando se usa la paleta por defecto, las aplicaciones pueden usar valores de color para especificar el color de plumas o texto. Si el color requerido no se encuentra en la paleta, Windows aproxima el color usando el más parecido de la paleta. Si una aplicación pide un pincel sólido de un color que no está en la paleta, Windows simula el color mediante mezcla de pixels con colores presentes en la paleta.

Para evitar aproximaciones y mezclas de pixels, se pueden especificar colores para plumas, pinceles y texto usando índices de

paleta de colores en lugar de valores de colores. Un índice de paleta de colores es un entero que identifica una entrada en una paleta específica. Se pueden usar índices de paleta en lugar de valores de color, pero se debe usar la macro [PALETTEINDEX](#) para crearlos.

Los índices de paleta sólo se pueden usar en dispositivos que soporten paletas de color. Esto puede hacer que nuestros programas sean dependientes del dispositivo, ya que no podremos usar índices para cualquier dispositivo. Para evitarlo, cuando se usa el mismo código para dibujar tanto en dispositivos con o sin paleta, se deben usar valores de color relativos a la paleta para especificar colores de plumas, pinceles o texto. Estos valores serán idénticos a los valores de colores, excepto cuando se crean pinceles sólidos.

En dispositivos con paleta, un color de pincel sólido especificado por un valor de color relativo a la paleta está sujeto a aproximación de color en lugar de a mezcla de pixels. Para crear valores de color relativos a la paleta se debe usar la macro [PALETTERGB](#).

Windows no permite cambiar las entradas de la paleta por defecto. Si se quiere usar otros colores en lugar de los que ésta contiene, se debe crear una paleta lógica propia y seleccionarla en el contexto de dispositivo.

## Paleta lógica

Una paleta lógica es una paleta que crea una aplicación y que se asocia con un contexto de dispositivo dado.

Para crear una paleta lógica se usa la función [CreatePalette](#). Antes es necesario llenar la estructura [LOGPALETTE](#), que especifica el número de entradas en la paleta y el valor de cada una, después se debe pasar esa estructura a la función **CreatePalette**. La función devuelve un manipulador de paleta que puede usarse en otras operaciones para identificar la paleta.

```
#define NUMCOLORES 18
```

```

...
    PALETTEENTRY Color[NUMCOLORES] = {
        //peRed, peGreen, peBlue, peFlags
        {0,0,0,PC_NOCOLLAPSE,
        {0,20,0,PC_NOCOLLAPSE,
...
        ;
    LOGPALETTE *logPaleta;
...
        // Crear una paleta nueva:
        logPaleta =
        (LOGPALETTE*)malloc(sizeof(LOGPALETTE) +
            sizeof(PALETTEENTRY) * NUMCOLORES);
        if(!logPaleta)
            MessageBox(hwnd, "No pude bloquear la memoria
de la paleta",
                "Error", MB_OK);

        logPaleta->palVersion = 0x300;
        logPaleta->palNumEntries = NUMCOLORES;
        for(i = 0; i < NUMCOLORES; i++) {
            logPaleta->palPalEntry[i].peBlue =
Color[i].peBlue;
            logPaleta->palPalEntry[i].peGreen =
Color[i].peGreen;
            logPaleta->palPalEntry[i].peRed =
Color[i].peRed;
            logPaleta->palPalEntry[i].peFlags =
Color[i].peFlags;

            hPaleta = CreatePalette(logPaleta);
            if(!hPaleta)
                MessageBox(hwnd, "No pude crear la paleta",
"Error", MB_OK);
            free(logPaleta);

```

Para usar los colores de una paleta lógica, la aplicación selecciona la paleta dentro de un contexto de dispositivo, usando la función [SelectPalette](#). Los colores de la paleta estarán disponibles tan pronto como la paleta sea activada.

```

case WM_PAINT:

```

```
hDC = BeginPaint(hwnd, &ps);
hPaletaOld = SelectPalette(hDC, hPaleta, FALSE);
hBrush = CreateSolidBrush(PALETTEINDEX(6));
hOldBrush = (HBRUSH)SelectObject(hDC, hBrush);
Rectangle(hDC, 20, 20, 40, 40);
SelectObject(hDC, hOldBrush);
DeleteObject(hBrush);
SelectPalette(hDC, hPaletaOld, FALSE);
EndPaint(hwnd, &ps);
break;
```

Es posible limitar el tamaño de las paletas lógicas para permitir las entradas que representen sólo los colores necesarios. Además, no se pueden crear paletas lógicas más grandes que el tamaño máximo de paleta, y ese valor depende del dispositivo. Para obtener el tamaño máximo de la paleta se usa la función [GetDeviceCaps](#) para recuperar el valor del valor [SIZEPALETTE](#).

Si bien es posible especificar cualquier color para una entrada de una paleta lógica, es posible que no todos los colores puedan ser generados por el dispositivo. Windows no proporciona una forma de averiguar qué colores son soportados, pero la aplicación puede descubrir el número total de esos colores leyendo la resolución de color del dispositivo. La resolución de color, especificada en bits de colores por pixel, es igual al valor [COLORRES](#) revuelto por la función [GetDeviceCaps](#). Un dispositivo que tenga una resolución de color de 18 tiene 262.144 colores posibles. Si una aplicación pide un color no soportado, Windows elige una aproximación apropiada.

Una vez que una paleta lógica a sido creada, se pueden cambiar colores dentro de ella usando la función [SetPaletteEntries](#). Si la paleta lógica ha sido seleccionada y activada, los cambios no afectarán inmediatamente a los colores actualmente mostrados. Para eso es necesario usar las funciones [UnrealizeObject](#) y [RealizePalette](#), de ese modo los colores de la pantalla se actualizarán. En algunos casos, puede ser necesario deseleccionar, desactivar, seleccionar y activar una paleta lógica para asegurarse de que los colores se actualizarán exactamente como se requiere. Si se selecciona una paleta lógica en más de un contexto de

dispositivo, los cambios en la paleta lógica afectan a todos los contextos de dispositivo en las que ha sido seleccionada.

Se puede cambiar el número de entradas de una paleta lógica usando la función [ResizePalette](#). Si la aplicación reduce el tamaño, las entradas que quedan permanecen sin cambios. Si se aumenta el tamaño, Windows asigna los colores para cada nueva entrada a negro (0, 0, 0) y el banderín a cero.

También es posible recuperar los valores del color y del banderín para entradas en una paleta lógicas dada mediante la función [GetPaletteEntries](#). Se puede recuperar el índice para una entrada dentro de una paleta lógica que coincida lo más cercanamente posible con un color especificado usando la función [GetNearestPaletteIndex](#).

Cuando no se necesite más una paleta lógica, se debe eliminar usando la función [DeleteObject](#). Hay que asegurarse de que la paleta lógica no permanece seleccionada dentro de un contexto de dispositivo antes de borrarla.

```
case WM_DESTROY:
    DeleteObject(hPaleta);
    PostQuitMessage(0);
    break;
```

## Paleta de sistema

Windows mantiene una paleta de sistema para cada dispositivo que use paletas. Esta paleta contiene los valores de todos los colores que pueden ser mostrados o usados actualmente por el dispositivo. Las aplicaciones no tienen acceso a la paleta de sistema directamente, al contrario, Windows tiene control absoluto de la paleta del sistema y permite el acceso sólo a través de las paletas lógicas.

Para ver el contenido de la paleta del sistema se usa la función [GetSystemPaletteEntries](#). Esta función recupera el contenido de una o más entradas, hasta el número total de entradas en la paleta de sistema. El número total es siempre el mismo que el devuelto para el valor [SIZEPALETTE](#) de la función [GetDeviceCaps](#) y es el mismo que el tamaño máximo de cualquier paleta lógica dada.

Ya hemos dicho que no es posible modificar los colores de la paleta de sistema directamente, sino sólo cuando se activan paletas lógicas. Antes de activar una paleta, Windows examina cada color requerido e intenta encontrar una entrada en la paleta del sistema que coincida exactamente. Si Windows encuentra ese color, asigna el índice de la paleta lógica para que corresponda con ese índice de la paleta del sistema. Si no lo encuentra, se copia el color requerido en una entrada no usada de la paleta de sistema antes de asignar el índice. Si todas las entradas de la paleta de sistema están en uso, Windows asigna al índice de la paleta lógica la entrada de la paleta de sistema cuyo color sea lo más parecido posible al color requerido. Una vez que los índices han sido asignados, no es posible ignorarlos. Por ejemplo, no es posible usar índices de la paleta de sistema para especificar colores, sólo se permite el uso de índices de la paleta lógica.

Se puede modificar el modo en que los índices serán asignados cuando se seleccionen los valores de la paleta lógica mediante el miembro `peFlags` de la estructura [PALETTEENTRY](#). Por ejemplo, el banderín [PC\\_NOCOLLAPSE](#) indica a Windows que copie inmediatamente el color requerido en una entrada no usada de la paleta de sistema aunque la paleta de sistema ya contenga ese color. Además, el flag `PC_EXPLICIT` indica a Windows que le asigne al índice de paleta lógica un índice explícito de la paleta de sistema. (Para eso se debe dar el índice de la paleta de sistema en la palabra de menor orden de la estructura [PALETTEENTRY](#)).

Las paletas pueden ser activadas como paleta de fondo o como paleta de primer plano especificando [TRUE](#) o [FALSE](#) para el parámetro `bForceBackground` en la función [SelectPalette](#),

respectivamente. Sólo puede existir una paleta de primer plano en el sistema al mismo tiempo. Si una ventana o una descendiente suya es la activa actualmente, puede activar una paleta de primer plano. En caso contrario la paleta se activa como paleta de fondo, independientemente del valor del parámetro `bForceBackground`. La principal propiedad de una paleta de primer plano es que cuando se activa, puede sobrescribir todas las entradas de la paleta de sistema (excepto las estáticas). Windows lo permite marcando las entradas de la paleta de sistema que no sean estáticas como no usadas antes de activar una paleta de primer plano, pudiendo eliminarse todas las entradas usadas. La paleta de primer plano usa todos los colores no estáticos posibles. Las de fondo sólo pueden usar las que permanezcan libres y que estén disponibles para el primero que las solicite. Normalmente, se usan paletas de fondo con ventanas hijas que activen sus propias paletas. Esto ayuda a minimizar el número de cambios en la paleta de sistema.

Una entrada de paleta de sistema no usada es cualquiera que no está reservada y que no contenga un color estático.

En estos tiempos de potentes tarjetas gráficas, con millones de colores y enormes velocidades de proceso, este tipo de preocupaciones ha pasado (felizmente) a la historia. No hace muchos años, muchas tarjetas gráficas limitaban sus paletas a 16 ó 256 colores, aunque fueran capaces de mostrar muchos más. El resultado es que frecuentemente distintas aplicaciones activaban diferentes paletas. Sólo la aplicación que tiene el foco puede activar una paleta de primer plano, de modo que las aplicaciones que perdían el foco también podían ver modificados gran parte de sus colores. El resultado es que los colores de las aplicaciones cambiaban continuamente al cambiar de aplicación, creando un efecto molesto y poco estético.

Las entradas reservadas están marcadas explícitamente con el valor `PC_RESERVED`. Esas entradas se crean cuando se activan paletas lógicas para animaciones de paleta. Las entradas de color estáticas se crean por Windows y corresponden a los colores de la

paleta por defecto. Se puede usar la función [GetDeviceCaps](#) para recuperar el valor [NUMRESERVED](#), que especifica el número de entradas de la paleta de sistema reservados para colores estáticos.

Ya que la paleta de sistema tiene un número de entradas limitado, la selección y activación de una paleta lógica para un dispositivo dado puede afectar a los colores asociados con otras paletas lógicas del mismo dispositivo.

Esos cambios de color son especialmente dramáticos cuando se dan en una pantalla. Para asegurarse de que se usan los colores de una paleta lógica seleccionada del modo más fiel hay que resetear la paleta antes de usarla. Esto se hace llamando a las funciones [UnrealizeObject](#) y [RealizePalette](#). Usando estas funciones se obliga a Windows a reasignar los colores de la paleta lógica a colores adecuados de la paleta de sistema.

## Ejemplo 20

**Nota:**

Este ejemplo funcionará con configuraciones de pantalla de 256 colores o menos, de un modo diferente a si existen más colores. En el caso de pantallas con 256 o menos colores, la paleta creada usará mezclas de pixels.



# Capítulo 23 Objetos básicos del GDI: El Mapa de Bits (Bitmap)

Windows usa mapas de bits para muchas cosas. Si te fijas en la ventana de tu aplicación, (ahora probablemente sea un explorador de Internet), verás unos cuantos. Por ejemplo, en la barra de herramientas. Pero también son mapas de bits las flechas de las barras de desplazamiento, los dibujos de los botones de cerrar, maximizar o minimizar, los iconos, etc.

Simplificando, podemos considerar que un mapa de bits es un rectángulo de pixels que en conjunto forman una imagen.

Dentro del API un mapa de bits es una colección de estructuras: una cabecera que contiene información sobre dimensiones, tamaño del array de bits, etc. Una paleta lógica y un array de bits, que relacionan los pixels con la paleta.

## Tipos de mapas de bits

Windows trabaja con dos tipos de mapas de bits: dependientes del dispositivo (DDBs) e independientes del dispositivo (DIBs).

Los DDBs provienen de las primeras versiones de Windows, anteriores a la 3.0. Más adelante, debido a ciertos problemas con los dispositivos, se crearon los DIBs.

Pero por el momento esto no nos preocupa mucho, ya lo veremos en profundidad. Lo que nos interesa ahora es cómo usar mapas de bits en nuestros programas, y en eso nos vamos a centrar.

## Crear un mapa de bits

Existen funciones para crear mapas de bits, pero están pensadas para generarlos de una forma más o menos dinámica, más que para usar mapas de bits que contengan imágenes ya existentes.

En el presente capítulo nos limitaremos a usar mapas de bits que ya existan en forma de fichero ".bmp". Nuestros programas pueden trabajar con esos ficheros de dos formas: mapas de bits en ficheros de recursos, o mapas de bits almacenados en ficheros en disco.

## Fichero de recursos

Esta es la forma de añadir mapas de bits que la aplicación pueda usar como parte de controles, por ejemplo, mapas de bits en menús o botones, pero esos mapas de bits pueden usarse por la aplicación para cualquier otra cosa. Hay que tener en cuenta que esos mapas de bits se incluyen en el ejecutable, por lo tanto no es muy recomendable usar mapas de bits muy grandes, ya que eso aumentará considerablemente el tamaño del fichero ".exe".

El modo de usar estos recursos es añadir una línea **BITMAP** en nuestro fichero de recursos:

```
#include <windows.h>
#define MASCARA 1000
#define CM_RECURSO 100
#define CM_FICHERO 101

Icono ICON "Food.ico"

Bitmap BITMAP "meninas24.bmp"
MASCARA BITMAP "mascara24.bmp"

Menu MENU
BEGIN
    POPUP "Opciones"
    BEGIN
        MENUITEM "&Bitmap de recurso", CM_RECURSO
        MENUITEM "&Bitmap de fichero", CM_FICHERO
```

```
END  
END
```

Esto es una parte del proceso, la otra parte consiste en obtener un manipulador de mapa de bits en nuestra aplicación, para ello se usa la función [LoadBitmap](#). Esta función requiere dos parámetros, el primero es un manipulador de la instancia que contiene el mapa de bits. Generalmente será la instancia actual, pero los recursos, como veremos en el futuro, también pueden estar contenidos en bibliotecas dinámicas. El segundo parámetro es el identificador del recurso en forma de cadena, o (si hemos usado un entero) el resultado de aplicar la macro [MAKEINTRESOURCE](#) a ese entero.

Por supuesto, una vez que no necesitamos el manipulador, debemos eliminarlo usando la función [DeleteObject](#).

```
static HINSTANCE hInstance;  
static HBITMAP hBitmapRes;  
static HBITMAP hMascara;  
  
...  
switch (msg) /* manipulador del mensaje  
*/  
{  
    case WM_CREATE:  
        hInstance = ((LPCREATESTRUCT)lParam)->hInstance;  
        hBitmapRes = LoadBitmap(hInstance, "Bitmap");  
        hMascara = LoadBitmap(hInstance,  
MAKEINTRESOURCE(MASCARA));  
    ...  
    case WM_DESTROY:  
        DeleteObject(hBitmapRes);  
        DeleteObject(hMascara);  
        PostQuitMessage(0); /* envía un mensaje  
WM_QUIT a la cola de mensajes */  
        break;
```

## Fichero BMP

Otro modo de obtener mapas de bits para usarlos en nuestra aplicación es directamente a partir de ficheros BMP, sin necesidad de fichero de recursos. Para leer uno de esos ficheros se usa la función [LoadImage](#).

En realidad, la función [LoadImage](#) se puede usar en ambos casos, pero considero que la función [LoadBitmap](#) es más cómoda para obtener un manipulador de mapa de bits cuando se trata de recursos.

Necesitamos indicar como manipulador de instancia el valor [NULL](#), si se usa un valor de instancia, generalmente se usará para obtener un mapa de bits de recursos. El segundo parámetro es el nombre del fichero, o el nombre del recurso. El tercer parámetro indica el tipo de imagen, que puede ser un mapa de bits, un icono o un cursor. Los dos siguientes indican un tamaño de imagen, pero no se usan cuando se trata de mapas de bits. El último parámetro permite ajustar algunas opciones, pero sobre todo nos interesa el valor [LR\\_LOADFROMFILE](#).

```
static HBITMAP hBitmapRes;
...
switch (msg)                /* manipulador del mensaje
*/
{
    case WM_CREATE:
        hInstance = (LPCREATESTRUCT)lParam->hInstance;
        hBitmapFil = (HBITMAP)LoadImage(NULL,
"Abanicos.bmp", IMAGE_BITMAP,
        0, 0, LR_LOADFROMFILE);
    ...
    case WM_DESTROY:
        DeleteObject(hBitmapFil);
        PostQuitMessage(0);    /* envía un mensaje
WM_QUIT a la cola de mensajes */
        break;
```

Por supuesto, cuando ya no se necesite, se debe liberar el recurso usando [DeleteObject](#).

## Mostrar un mapa de bits

Al contrario que con otros objetos del GDI que hemos manejado hasta ahora, los mapas de bits no se seleccionan directamente en un DC de ventana. Esto se debe a que Windows sólo permite seleccionar los mapas de bits en DC de memoria, y además, sólo en uno al mismo tiempo.

Por lo tanto, para mostrar un mapa de bits tendremos que realizar algunas tareas:

1. Crear un DC de memoria.
2. Seleccionar el mapa de bits en ese DC.
3. Usar una de las funciones disponibles para mostrar el mapa de bits.
4. Borrar el DC de memoria.

El DC de memoria que necesitamos no es un DC cualquiera, debe tratarse de un DC compatible con el dispositivo en el que queramos mostrar el mapa de bits. Para crear uno de esos DCs se usa la función [CreateCompatibleDC](#).

Seleccionar el mapa de bits es una operación similar a seleccionar pinceles, brochas o paletas. Se usa la función [SelectObject](#), el mapa de bits se maneja mediante un manipulador de mapas de bits, un [HBITMAP](#), que habremos obtenido de uno de los modos que hemos explicado más arriba.

En cuanto a las funciones que podemos usar para mostrar mapas de bits, la más sencilla es [BitBlt](#). Pero hay otras, aunque algunas no están disponibles en todas las versiones de Windows.

Finalmente liberamos el DC usando la función [DeleteDC](#).

## Funciones de visualización de mapas de bits

Las funciones que veremos de momento son:

## BitBlt

Muestra un mapa de bits, sin escalar ni deformar. Se especifica un rectángulo de destino, si el mapa de bits origen es más pequeño se rellena con el color de fondo, si es más grande, se recorta.

No es necesario mostrar todo el mapa de bits, podemos mostrar sólo un área rectangular del tamaño que queramos, y a partir de la dirección que prefiramos.

Por ejemplo, es muy frecuente crear un único mapa de bits con muchos gráficos reunidos, formando una lista o un mosaico. La función [BitBlt](#) nos permite mostrar sólo uno de esos gráficos, sin preocuparnos por el resto:

```
void CTrozo(HDC hDC, HWND hWnd, HBITMAP hBitmap)
{
    HDC memDC;

    memDC = CreateCompatibleDC(hDC);
    SelectObject(memDC, hBitmap);
    BitBlt(hDC, 135, 225, 40, 40, memDC, 135, 225, SRCCOPY);
    DeleteDC(memDC);
}
```

Esta función muestra en el rectángulo de la ventana que empieza en las coordenadas (135, 225), y que tiene 40x40 pixels, el trozo del mapa de bits que empieza en las coordenadas (135, 225). Como el mapa de bits se muestra en relación de un pixel por cada pixel del mapa de bits, no habrá distorsión, el trozo de mapa de bits mostrado tendrá el mismo tamaño que el rectángulo en el que se muestra.

## StretchBlt

Muestra un mapa de bits o parte de él. El mapa se adapta al rectángulo especificado como destino, estirándose en cada

dirección de la forma necesaria para ocuparlo por completo. El modo de estiramiento se puede modificar mediante la función [SetStretchBltMode](#).

Al igual que en el caso anterior, es posible mostrar un trozo del mapa de bits, pero con esta función podemos escalar ese trozo para adaptarlo a la superficie rectangular que queramos:

```
void CAMpliacion(HDC hDC, HWND hWnd, HBITMAP hBitmap)
{
    HDC memDC;
    BITMAP bm;
    RECT re;

    memDC = CreateCompatibleDC(hDC);
    SelectObject(memDC, hBitmap);
    GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&bm);
    GetClientRect(hWnd, &re);
    StretchBlt(hDC, 0, 0, re.right, re.bottom, memDC,
               135, 225, 40, 40, SRCCOPY);
    DeleteDC(memDC);
}
```

En este caso se muestra el mismo trozo del mapa de bits que con el ejemplo anterior, pero en lugar de usar el mismo tamaño en la pantalla, el trozo se estirará para adaptarse al área indicada, que no es otra que toda el área de cliente.

Para indicar toda el área de cliente hemos obtenido sus coordenadas mediante la función [GetClientRect](#).

Para obtener el tamaño del mapa de bits usamos la función [GetObject](#) que nos devuelve una estructura [BITMAP](#) asociada al mapa de bits.

## PlgBlt (Sólo en Windows NT)

Muestra el mapa de bits o parte de él, deformándolo para adaptarlo al paralelogramo indicado mediante un array de tres puntos.

**Nota:**

Un paralelogramo queda definido por tres puntos, ya que el cuarto se puede calcular a partir de los otros tres. Esto es debido a que los paralelogramos son cuadriláteros con sus lados paralelos dos a dos.

La imagen resultante parece haber sido proyectada sobre una pantalla no paralela al observador.

En lo demás, la función es parecida a [StretchBlt](#). También requiere que se defina el área del mapa de bits que se mostrará, permitiendo mostrar sólo una parte.

Además, se puede especificar una máscara opcional. La máscara es un mapa de bits monocromo, en la que los pixels de valor uno indican que se debe copiar el pixel, y los de valor cero indican que se debe conservar el fondo.

## **MaskBlt (Sólo en Windows NT)**

Esta es la más complicada de las funciones para mostrar mapas de bits. Funciona de modo análogo a [BitBlt](#), en lo que respecta a que se mantiene la relación de pixels uno a uno, sin deformar la imagen. Pero usa un segundo mapa de bits monocromo como máscara. Esa máscara se puede combinar mediante códigos ROP el mapa de bits con la imagen actual de la ventana y con el pincel actual seleccionado en el contexto de dispositivo.

Para esta función se deben especificar dos mapas de bits. El primero es la imagen a mostrar, el segundo es un mapa de bits monocromo que define una máscara.

Además debemos indicar dos códigos ROP ternarios, uno de los cuales se usa para el fondo y el otro para el primer plano. Si un pixel del mapa de bits de la máscara es un uno, se aplicará el código ROP para el primer plano, si es cero, el del fondo.



Esto nos permite mostrar mapas de bits con formas no rectangulares, con una gran libertad a la hora de relacionar la imagen actualmente en pantalla con la nueva.

## Códigos ROP ternarios

Existen quince valores posibles, que agruparemos según su función.

El primer grupo lo componen los códigos ROP en los que el segundo bitmap no tiene importancia ya que no se usa para obtener el resultado final:

- **BLACKNESS**: rellena el área indicada con el color asociado al índice 0 de la paleta física. Este color es negro en la paleta física por defecto.
- **DSTINVERT**: invierte el área de destino. Invertir significa, aplicar el operador de bits NOT para cada valor de color pixel. El resultado en pantalla es un negativo fotográfico.
- **WHITENESS**: Rellena el rectángulo de destino usando el color asociado al índice 1 de la paleta física. (Este color es blanco para la paleta física por defecto.)

Hay dos códigos que implican al pincel asociado actualmente al contexto de dispositivo, que puede ser de cualquier tipo, y no necesariamente un creado a partir de un mapa de bits:

- **PATCOPY**: rellena el área indicada con el pincel actual.
- **PATINVERT**: combina los colores del pincel actual con los colores en el rectángulo destino usando el operador booleano XOR.

El resto de los códigos ROP implican un segundo mapa de bits:

- **MERGECOPY**: mezcla los colores en el rectángulo de origen con el patrón especificado usando la operación booleana AND.

- **MERGEPAINT**: mezcla los colores invertidos del rectángulo de origen con los colores del rectángulo de destino usando la operación booleana OR.
- **NOTSRCCOPY**: copia el rectángulo de origen invertido al rectángulo de destino.
- **NOTSRCERASE**: combina los colores de los rectángulos de origen y destino usando el operador booleano OR y después invierte el color resultante.
- **PATPAINT**: combina los colores del patrón con los colores invertidos del rectángulo de origen usando el operador booleano OR. El resultado de esta operación se combina con los colores del rectángulo de destino usando el operador booleano OR.
- **SRCAND**: combina los colores de los rectángulos de origen y destino usando el operador booleano AND.
- **SRCCOPY**: copia el rectángulo de origen directamente en el rectángulo de destino.
- **SRCERASE**: combina los colores invertidos el rectángulo de destino con los colores del rectángulo de origen usando el operador booleano AND.
- **SRCINVERT**: combina los colores de los rectángulos de origen y destino usando el operador booleano XOR.
- **SRCPAINT**: combina los colores de los rectángulos de origen y destino usando el operador booleano OR.

## Códigos ROP cuádruples

La función **MaskBlt** requiere como parámetro un código ROP cuádruple, estos códigos se obtienen combinando dos códigos ROP ternarios mediante la macro **MAKEROP4**.

## Pinceles creados a partir de mapas de bits

Es posible crear pinceles de tramas basadas en mapas de bits, como adelantamos en el **capítulo 20**. Para hacerlo se usa la función **CreatePatternBrush**.

```
HBRUSH pincel, anterior;
HBITMAP hBitmapLazo;

hBitmapLazo = LoadBitmap(hInstance, "Lazo");
pincel = CreatePatternBrush(lazo);
anterior = SelectObject(hDC, pincel);
...
SelectObject(hDC, anterior);
DeleteObject(pincel);
DeleteObject(hBitmapLazo);
```

Existen además dos funciones para rellenar superficies trabajando con pinceles.

## PatBlt

Sirve para rellenar un área rectangular usando el pincel actual. Además, se puede especificar un código ROP para indicar el modo en que deben combinarse el pincel con el fondo. No todos los código ROP se pueden usar con esta función, solo:

ROP	Descripción
PATCOPY	Copia el patrón al mapa de bits de destino.
PATINVERT	Combina el mapa de bits de destino con el patrón usando el operador OR.
DSTINVERT	Invierte el mapa de bits de destino.
BLACKNESS	Pone todas las salidas a ceros.
WHITENESS	Pone todas las salidas a unos.

## ExtFloodFill

También sirve para rellenar áreas usando el pincel actual, pero [ExtFloodFill](#) permite que el área a rellenar sea irregular. Hay que indicar el punto donde se empieza a rellenar y el color del recinto que delimita el área. No se pueden especificar códigos ROP.

#### Nota:

Existe una función [FloodFill](#) sirve para lo mismo que [ExtFloodFill](#), pero proviene de versiones anteriores del API, y actualmente se considera obsoleta, tan sólo se mantiene por compatibilidad.

## Estructuras de datos

Existen varias estructuras de datos relacionadas con los mapas de bits, aunque la mayoría están relacionadas con paletas o con el modo de almacenar los datos que contienen o son muy específicos de mapas de bits independientes del dispositivo. Sin embargo hay una estructura que nos resultará muy útil:

### BITMAP

La estructura es:

```
typedef struct tagBITMAP {    // bm
    LONG    bmType;
    LONG    bmWidth;
    LONG    bmHeight;
    LONG    bmWidthBytes;
    WORD    bmPlanes;
    WORD    bmBitsPixel;
    LPVOID  bmBits;
} BITMAP;
```

Los datos que más nos interesan son *bmWidth* y *bmHeight*, que indican las dimensiones de anchura y altura, respectivamente, del mapa de bits. El resto de los datos, al menos de momento, no nos interesan.

Para obtener los datos de esta estructura para un mapa de bits concreto, se usa la función [GetObject](#), por ejemplo:

```
HBITMAP hBitmap;  
BITMAP bm;  
  
GetObject(hBitmap, sizeof(BITMAP), (LPSTR) &bm);
```

El primer parámetro es el manipulador del mapa de bits que nos interesa, el segundo el tamaño de la estructura **BITMAP** y el tercero un puntero a la estructura que recibirá los datos.

## Modos de estiramiento (stretch modes)

Vimos antes, al hablar de la función **StretchBlt** que existen varios modos diferentes de estiramiento, o mejor dicho, de estrechamiento, ya que estos modos afectan al mapa de bits mostrado cuando se pierden puntos. En concreto existen cuatro, (los otros cuatro son equivalentes para Windows 95), comentaremos algo sobre ellos:

- **BLACKONWHITE**: cuando se pierden puntos, se agrupan realizando una operación booleana AND entre los pixels eliminados. Si el mapa de bits es monocromo, este modo conserva los pixels negros a costa de los blancos.
- **COLORONCOLOR**: los puntos perdidos no se tienen en cuenta. Este modo borra todas las líneas de pixels que no se visualizan, sin intentar preservar su información.
- **HALFTONE**: proyecta los pixels del mapa de bits en el rectángulo de destino, el color medio de los pixels que resultan agrupados se aproxima de este modo al original. Si se activa este modo, hay que usar la función **SetBrushOrgEx** para cambiar el origen del pincel. Si se falla al hacerlo, habrá un desalineamiento del pincel.
- **WHITEONBLACK**: cuando se pierden puntos, se agrupan realizando una operación booleana OR entre los pixels eliminados. Si el mapa de bits es monocromo, este modo conserva los pixels blancos a costa de los negros.

Los modos **BLACKONWHITE** y **WHITEONBLACK** se usan sobre todo con mapas de bits monocromo, los otros dos con mapas de bits en color. El modo **HALFTONE** proporciona mucho mejor resultado, pero es mucho más lento, además, requiere usar la función [SetBrushOrgEx](#).

También existen dos funciones relacionadas con estos modos: [GetStretchBltMode](#) para averiguar el modo actual de un contexto de dispositivo, y [SetStretchBltMode](#) para cambiarlo.

## Mapas de bits de stock

Aunque no es probable que nos resulten útiles, también existen mapas de bits de stock, que son los que se usan para las barras de deslizamiento y los botones de minimizar, maximizar, etc.

Podemos obtener esos mapas de bits usando las funciones [LoadBitmap](#) y [LoadImage](#), usando **NULL** como manipulador de instancia y uno de los identificadores especiales para los mapas de bits:

```
HDC memDC;
BITMAP bm;
HBITMAP hBitmap;
RECT re;

memDC = CreateCompatibleDC(hDC);
hBitmap = LoadBitmap(NULL, MAKEINTRESOURCE(OBM_CLOSE));
SelectObject(memDC, hBitmap);
GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&bm);
BitBlt(hDC, 10, 10, bm.bmWidth, bm.bmHeight, memDC, 0,
0, SRCCOPY);
DeleteObject(hBitmap);
DeleteDC(memDC);
```

## Ejemplo 21

**Nota:**

Algunas de las funciones usadas en este ejemplo sólo funcionan en Windows NT o versiones superiores. El mayor tamaño de este ejemplo se debe a que incluye los ficheros de mapas de bits de ejemplo.

# Capítulo 24 Objetos básicos del GDI: La Fuente (Font)

Llegamos por fin a un objeto básico que se seguro que necesitamos en casi todos nuestros programas, y que estarías echando de menos: el texto.

Veremos en este capítulo que si bien, mostrar texto en pantalla es fácil, (basta con una función del API), el tema no es tan sencillo como pudiera pensarse, ya que Windows nos ofrece muchas posibilidades a la hora de trabajar con texto. Podemos elegir la forma, el tamaño, orientación, estilo, espaciado... de cada fuente. En este capítulo aprenderemos a mostrar texto y también a personalizarlo.

## Mostrar un texto simple

El API siempre dispone de objetos de stock, de modo que si sólo queremos mostrar un texto en nuestra ventana, podemos usar la fuente por defecto para ello, sin complicarnos la vida.

La función más simple para mostrar texto es [TextOut](#), que usa la fuente activa.

La forma de usarla es tan sencilla como indicar un manipulador de contexto de dispositivo, las coordenadas de inicio del texto, el propio texto, y el número de caracteres a mostrar. Por supuesto, seguiremos usando el mensaje [WM\\_PAINT](#) para actualizar la pantalla:

```
case WM_PAINT:
    hDC = BeginPaint(hwnd, &ps);
    TextOut(hDC, 10, 10, "Hola, mundo!", 12);
```



```
23);      TextOut(hDC, 20, 30, "Curso WinAPI con Clase.",
           EndPaint(hwnd, &ps);
           break;
```

## Cambiar el color del texto

El texto mostrado por el ejemplo anterior usa la fuente del sistema, con el color por defecto: letras negras sobre fondo blanco.

Esto no resulta muy elegante, que digamos. Es bastante rudimentario y, desde luego, no es lo mejor que sabremos hacer.

Podemos cambiar el color del fondo, usando la función que ya conocemos [SetBkColor](#), o podemos evitar el parche de color correspondiente al fondo, de modo que las letras se muestren sobre el contenido actual del fondo, sea del color que sea. Para lograr esto bastará con activar el modo transparente para el fondo, con la función que ya hemos usado: [SetBkMode](#).

```
SetBkColor(hDC, RGB(40,40,240));
SetBkMode(hDC, TRANSPARENT);
```

El valor por defecto para el modo del fondo es opaco, y para activarlo se usa el valor [OPAQUE](#).

El siguiente paso es modificar el color del texto, para ello disponemos de otra función del API: [SetTextColor](#), que precisa un manipulador de contexto de dispositivo y una referencia de color, para indicar el nuevo color del texto.

```
SetTextColor(hDC, RGB(255,0,0));
```

Si necesitásemos averiguar el color actual del texto, podemos usar la función [GetTextColor](#).

## Ejemplo 22

El siguiente ejemplo demuestra el modo más simple de mostrar texto:

### Crear fuentes personalizadas

El siguiente paso es aprender a usar cualquiera de las fuentes disponibles en el sistema, variando tanto el tipo, como el tamaño, orientación y estilo. Los parámetros que podemos cambiar en una fuente son muchos, de modo que intentaremos explicar cada uno de ellos para que nos sea más sencillo adaptar las fuentes a nuestros gustos o necesidades.

Para ello disponemos de dos funciones: [CreateFont](#) y [CreateFontIndirect](#). La primera precisa que le demos 14 parámetros que definen la fuente, la segunda crea la fuente a partir de los mismos parámetros, almacenados en una estructura [LOGFONT](#).

En ambos casos, al igual que ocurre con las fuentes de stock, recibiremos un manipulador de fuente [HFONT](#), que nos permitirá usar la fuente, ya sea para seleccionarla o eliminarla.

```
HFONT fuente;
HFONT grafico;
LOGFONT lf= {80, 0, 450, 450, 300, FALSE, FALSE, FALSE,
             DEFAULT_CHARSET, OUT_TT_PRECIS,
CLIP_DEFAULT_PRECIS,
             PROOF_QUALITY, DEFAULT_PITCH | FF_ROMAN,
"Webdings"};

fuente = CreateFont(-80, 0, 450, 450, 300, FALSE}, FALSE,
FALSE,
             DEFAULT_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
             PROOF_QUALITY, DEFAULT_PITCH | FF_ROMAN, "Times New
Roman");
grafico = CreateFontIndirect(&lf);
...
SelectObject(hDC, fuente);
```

```
...
SelectObject(hDC, grafico);
...
DeleteObject(fuente);
DeleteObject(grafico);
```

## Altura y anchura media de carácter

Existen algunos parámetros clave a la hora de medir una fuente, veamos algunos de ellos:

*Línea de base* es la línea sobre la que se apoyan las letras, en el caso de letras que sobresalen por debajo, la parte que apoya sobre esta línea es, generalmente, la del óvalo de la letra. La línea de base es lo que nos indica la inclinación de un texto.

*Punto* es la unidad de medida para fuentes. Un punto es, aproximadamente, 1/72 de pulgada, es decir, una fuente de 72 puntos de altura tendrá una pulgada de alto.

*Celda* es un rectángulo imaginario que contiene un carácter completo.

El tamaño de una fuente se define por dos parámetros: altura y anchura. Esto es evidente, ya que el texto tiene dos dimensiones. Sin embargo, aunque la altura es un parámetro claro, la anchura no lo es tanto. En algunas fuentes, los caracteres no tienen un ancho constante, sino proporcional a su forma, es decir, letras como la 'i' son más estrechas que otras, como la 'm'.

Estas fuentes, en la que cada carácter tiene una anchura diferente, son conocidas como de anchura proporcional; por contra, el otro tipo, en las que todos los caracteres tienen la misma anchura, se conoce como fuentes de anchura no proporcional:

Fuente no proporcional

Fuente proporcional

Fuentes

Es por eso que cuando hablamos de anchura de fuentes nos referimos a anchura media de carácter.

Los valores de altura son igualmente imprecisos, ya que existen caracteres de distintas alturas, por ejemplo, las letras como 'p', 'q' y 'g' sobresalen por la parte inferior, del mismo modo que las mayúsculas, o letras como la 't', 'l' y 'f' sobresalen por arriba. Normalmente se determina la altura como la distancia entre la parte inferior de la letra 'g' y la superior de la 'M'.



Alturas de fuentes

Es frecuente referirse también a la altura de la celda.

Cuando creamos fuentes, los valores de altura pueden ser positivos, negativos o cero. Los valores negativos indican medidas en función de la altura del carácter, y los positivos en función de la altura de la celda.

El valor nulo indica que se tome el valor por defecto para la altura.

En el caso de la anchura, el valor puede ser positivo o nulo. El valor positivo será el que se tome como anchura media, si es cero, se tomará el valor por defecto, que dependerá en cada caso del valor de altura elegido.

## El ángulo de escape

Cada carácter puede tener un ángulo sobre la línea de base. Podemos, por ejemplo, inclinar los caracteres 90° y escribir una línea horizontal:

⊥ ⊖ ⊗ ⊕ ⊙

Ángulo de escape

## El ángulo de orientación

Se refiere al ángulo formado por la línea de base con el eje x del dispositivo. Cuando creamos fuentes tenemos una precisión de décimas de grados para precisar dicho ángulo, de modo que un valor de 900 indica 90°.

## Peso

El peso indica cómo de gruesos son los trazos que se usan para mostrar el carácter, es lo que diferencia un carácter en negrita de uno normal. Tenemos mil valores posibles para el peso, los valores más bajos indican trazos finos, los más altos, trazos gruesos.

Como ayuda existen ciertas constantes predefinidas para asignar a este parámetro.

## Cursiva

Se trata de un banderín, si se activa (valor **TRUE**), se generará una fuente cursiva, si se desactiva, una fuente vertical.

## Subrayado

Otro banderín, si se activa se generará una fuente subrayada, en caso contrario, una normal.

## Tachado

Un tercer banderín, que si se activa genera una fuente tachada.

## Conjunto de caracteres

Permite elegir entre distintos conjuntos de caracteres. Recordemos que es ASCII no es el único conjunto existente, existen otros, y el API nos permite seleccionar algunos de ellos. Entre los más corrientes están, por ejemplo: el Windows, el Unicode<sup>tm</sup> y el de símbolos.

## Precisión de salida

Básicamente, nos permite elegir una fuente de dispositivo, matricial o TrueType, si existen varias de distinto tipo y el mismo

nombre.

## **Precisión de recorte**

Afecta al tipo de rotación de caracteres cuando cambiamos la orientación de la fuente.

## **Calidad**

Permite seleccionar, a la hora de mostrarla en pantalla, la calidad de la fuente.

## **Paso y familia**

Sirve para indicar un tipo alternativo de fuente cuando la seleccionada no está disponible o no se especifica una fuente concreta.

## **Nombre**

Nombre de la fuente seleccionada. Los ficheros de fuentes almacenan el aspecto visual de cada fuente. Esto nos da una enorme posibilidad a la hora de mostrar textos o símbolos.

## **Fuentes de stock**

AL igual que con otros objetos del GDI que hemos usado antes, en el caso de las fuentes también disponemos de seis fuentes de stock, que podemos seleccionar mediante la función [GetStockObject](#).

En concreto se trata de las siguientes:

<b>Valor</b>	<b>Significado</b>
<b>ANSI_FIXED_FONT</b>	Especifica una fuente de espacio no proporcional

basada en el conjunto de caracteres de Windows. Normalmente se usa una fuente Courier.

## ANSI\_VAR\_FONT

Especifica una fuente de espacio proporcional basada en el conjunto de caracteres de Windows. Normalmente se usa una fuente MS Sans Serif.

Especifica la fuente por defecto para el dispositivo especificado. Se trata, típicamente, de la fuente System para dispositivos de visualización. Para

## DEVICE\_DEFAULT\_FONT

algunas impresoras matriciales esta fuente es una que reside en la propia impresora. (Imprimir con esa fuente suele ser mucho más rápido que hacerlo con otra.).

## OEM\_FIXED\_FONT

Especifica una fuente de espacio no proporcional basada en un conjunto de caracteres OEM. Para ordenadores IBM® y compatibles, la fuente OEM está basada en el conjunto de caracteres del IBM PC.

## SYSTEM\_FONT

Especifica la fuente System. Es una fuente

de espacio proporcional basada en el conjunto de caracteres Windows, y se usa por el sistema operativo para mostrar los títulos de las ventanas, los nombres de menú y el texto en los cuadros de diálogo. La fuente System siempre está disponible. Otras fuentes sólo están disponibles si han sido instaladas.

Especifica una fuente de espacio no proporcional compatible con la fuente System en versiones de Windows anteriores a la 3.0.

## SYSTEM\_FIXED\_FONT

```
HFONT fuente, anterior;

fuente = GetStockObject(ANSI_FIXED_FONT);
anterior = SelectObject(hDC, fuente);
TextOut(hDC, 10, 10, "ANSI_FIXED_FONT", 15);

SelectObject(hDC, anterior);
DeleteObject(fuente);
```

## Alineamientos de texto

Cuando mostramos un texto usando la función [TextOut](#), [ExtTextOut](#) (que aún no hemos visto), indicamos unas coordenadas para situar el texto en el dispositivo. Estas coordenadas pueden referirse a diversos puntos dentro del texto. Podemos referirnos a la



línea de base, al centro del texto, a la esquina superior izquierda, a la esquina superior derecha, etc.

En concreto, tenemos las siguientes opciones:

- **TA\_BASELINE**: El punto de referencia es la línea de base del texto.
- **TA\_BOTTOM**: El punto de referencia es el borde inferior del rectángulo externo que contiene el texto.
- **TA\_TOP**: El punto de referencia es el borde superior del rectángulo que contiene el texto.
- **TA\_CENTER**: El punto de referencia se alinea horizontalmente con el centro del rectángulo que contiene el texto.
- **TA\_LEFT**: El punto de referencia es el borde izquierdo del rectángulo que contiene el texto.
- **TA\_RIGHT**: El punto de referencia es el borde derecho del rectángulo que contiene el texto.
- **TA\_NOUPDATECP**: El valor del cursor no se actualiza después de mostrar el texto.
- **TA\_UPDATECP**: El valor del cursor se actualiza después de mostrar el texto.

Pero no sólo eso, estas opciones se pueden combinar, aunque no de cualquier modo. Sólo se pueden agrupar valores tomando uno o ninguno de cada uno de los grupos:

- **TA\_LEFT**, **TA\_RIGHT** y **TA\_CENTER**
- **TA\_BOTTOM**, **TA\_TOP** y **TA\_BASELINE**
- **TA\_NOUPDATECP** y **TA\_UPDATECP**

Podemos, por ejemplo, combinar el valor **TA\_LEFT** con **TA\_BASELINE** o **TA\_CENTER** con **TA\_TOP**, etc. Para combinarlos se usa el operador de bits OR (|).

Para cambiar la alineación del texto se usa la función [SetTextAlign](#), para obtener el valor actual se usa [GetTextAlign](#).

Pero averiguar si uno de los bits está activo en el valor actual de alineamiento no es tan sencillo como pudiera parecer a primera

vista. Los valores `TA_xxx` son bits dentro de un valor de alineamiento, y como hemos visto esos bits pueden estar combinados entre ellos.

Si queremos comprobar si el valor de alineamiento actual contiene el valor `TA_LEFT` no bastará con comparar el valor de retorno de `GetTextAlign` con `TA_LEFT`, ya que el valor actual puede tener también los valores `TA_BOTTOM`, `TA_TOP`, `TA_BASELINE`, `TA_NOUPDATECP` o `TA_UPDATECP`. Tampoco bastará con un AND de bits, ya que no sabemos si, por ejemplo, `TA_CENTER` equivale a `TA_LEFT | TA_RIGHT`. (Podría ser).

Los pasos a seguir son:

1. Aplicar el operador de bits OR a los bits del grupo al que pertenece el valor que queremos verificar.
2. Aplicar el operador de bits AND entre el resultado anterior y al valor retornado por `GetTextAlign`.
3. Verificar la igualdad entre ese resultado y la bandera a verificar.

En nuestro ejemplo, tenemos:

1. `x = TA_LEFT | TA_RIGHT | TA_CENTER`
2. `y = x & GetTextAlign(hDC);`
3. Verificar si `y == TA_LEFT`

Si la alineación actual es, por ejemplo, `TA_RIGHT | TA_BASELINE`, y queremos comprobar si contiene el valor `TA_LEFT`, la sentencia C puede ser como esta:

```
if(TA_LEFT == ((TA_LEFT | TA_RIGHT | TA_CENTER) &
GetTextAlign(hDC)))...
```

## Separación de caracteres

Normalmente, cuando mostramos texto en un dispositivo, la separación entre caracteres está predeterminada, y depende del diseño de la fuente. Pero, con el fin de hacer que el texto sea más ancho, podemos aumentar la separación entre caracteres, usando la función [SetTextCharacterExtra](#).

Texto de prueba

T e x t o   d e   p r u e b a

Separación

Del mismo modo, podemos comprimir el texto usando valores de separación negativos.

Por último, podemos recuperar el valor de separación para un contexto de dispositivo determinado usando la función [GetTextCharacterExtra](#).

## Medidas de cadenas

A menudo nos interesa conocer las medidas que van a tener las cadenas en el dispositivo antes de mostrarlas, por ejemplo, para situarlas correctamente, separar las líneas adecuadamente, formatear párrafos, etc.

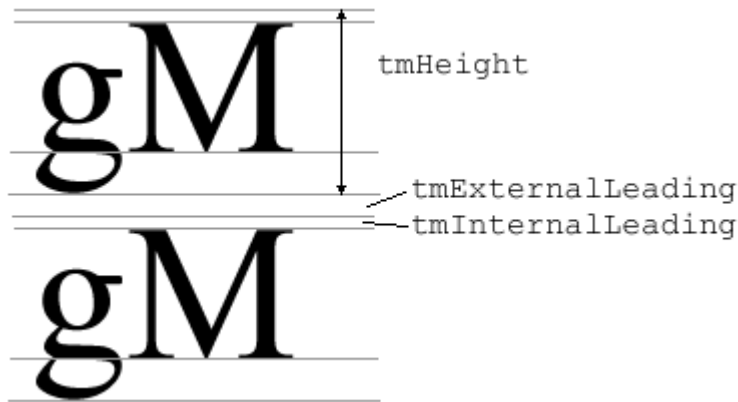
Disponemos de varias funciones en el API para esta tarea.

Empezaremos por la función [GetTextMetrics](#), que nos proporciona datos sobre la fuente actual de un contexto de dispositivo en una estructura [TEXTMETRIC](#).

Esta estructura nos informa principalmente sobre las medidas verticales de las líneas de texto, y nos da alguna información sobre medidas horizontales, aunque en este caso, algo menos precisas.

## Justificar texto

El API también nos proporciona funciones para mostrar texto justificado: que se ajusta a los márgenes derecho e izquierdo del área de visualización.



### Medidas de cadenas

Para ello deberemos usar dos funciones de forma conjunta. Por una parte

[GetTextExtentPoint32](#), que nos proporciona información sobre el tamaño de una línea de texto. Por otra, la

función [SetTextJustification](#), que prepara las cosas para que la siguiente función de salida de texto [TextOut](#) incluya la separación apropiada entre palabras.

La función [GetTextExtentPoint32](#) sirve para calcular el espacio necesario de pantalla necesario para mostrar una cadena. Esta información la podemos usar, por una parte, para averiguar si cierta cadena cabe en el espacio en que queremos mostrarla, y por otra, para saber cuanto espacio sobra, si es que cabe.

El espacio sobrante se debe repartir entre las palabras de la cadena a justificar, eso se hace mediante la función [SetTextJustification](#), que necesita como parámetros el manipulador del DC, el espacio extra, y el número de espacios entre palabras que contiene la cadena.

Hay que usar la función [SetTextJustification](#) con cuidado, ya que sus efectos permanecen hasta la siguiente llamada. Es decir, si usamos esta función para justificar una línea, y no anulamos su efecto, subsiguientes llamadas a [GetTextExtentPoint32](#) podrán falsear la medida de la cadena, ya que se usará más espacio entre palabras para calcular la longitud de la cadena.

Para anular el efecto de la función se usa la misma función, pero con un valor nulo en el segundo parámetro, y también en el tercero, aunque en realidad ese parámetro se ignora si el segundo es nulo.

```
    char *cadena = "Para ello deberemos usar dos funciones  
de";  
    SIZE tam;  
    RECT re;  
  
    GetClientRect(hwnd, &re); // Obtener tamaño de la  
ventana  
    SetTextJustification(hDC, 0, 0); // Anula cualquier  
justificación previa  
    GetTextExtentPoint32(hDC, cadena, strlen(cadena), &tam);  
// Calcular tamaño de cadena  
    SetTextJustification(hDC, re.right-tam.cx-20,  
espacios[i]); // Justificar  
    TextOut(hDC, 10, 10, cadena, strlen(cadena)); // Mostrar  
cadena
```

## Ejemplo 23

# Capítulo 25 Objetos básicos del GDI: Rectángulos y Regiones

## Rectángulos

Los rectángulos se usan en Windows para muchas cosas, y disponen de estructuras y funciones dedicadas a manejarlos.

Ya los hemos usado, por ejemplo, para invalidar parte de la ventana y obligar al Windows a actualizarla, para ello usamos la función [InvalidateRect](#), también los hemos usado para obtener las dimensiones del área de cliente con [GetClientRect](#).

Lo primero que tenemos que ver es la estructura [RECT](#), aunque ya la hemos usado muchas veces en este curso.

Esta estructura define un rectángulo mediante las coordenadas de la parte izquierda, superior, derecha e inferior del rectángulo: left, top, right y bottom. Podemos decir que estos valores definen dos puntos: la esquina superior izquierda y la inferior derecha. Es importante tener claro esto, ya que si la coordenada izquierda es mayor que la derecha, o la superior mayor que la inferior, no estaremos definiendo un rectángulo, al menos desde el punto de vista del API.

## Funciones para trabajar con rectángulos

Veremos a continuación algunas de las funciones para trabajar con rectángulos.

### Asignar rectángulos

En cuanto a las funciones de asignación, tenemos:

<b>Función</b>	<b>Utilidad</b>
<a href="#">SetRect</a>	Sirve para asignar valores a un rectángulo.
<a href="#">CopyRect</a>	Hace una copia de un rectángulo.
<a href="#">SetRectEmpty</a>	Crea un rectángulo vacío.

Un rectángulo vacío es aquel en el que todas sus coordenadas son cero.

## Comparaciones de rectángulos

Tenemos tres funciones para comparar rectángulos:

<b>Función</b>	<b>Utilidad</b>
<a href="#">IsRectEmpty</a>	Nos dice si un rectángulo es vacío.
<a href="#">EqualRect</a>	Nos dice si dos rectángulos son iguales.
<a href="#">PtInRect</a>	Nos dice si un punto está dentro de un rectángulo. Un punto en el lado inferior o en el derecho se considera fuera del rectángulo.

## Modificar rectángulos

Para modificar rectángulos, tenemos otras dos:

<b>Función</b>	<b>Utilidad</b>
<a href="#">InflateRect</a>	Nos permite agrandar o reducir un rectángulo.
<a href="#">OffsetRect</a>	Nos permite trasladar un rectángulo.

## Operaciones con rectángulos

Por último, para operar con rectángulos, tenemos otras tres:

<b>Función</b>	<b>Utilidad</b>
<a href="#">IntersectRect</a>	Obtiene la intersección de dos rectángulos.

**UnionRect** Obtiene la unión de dos rectángulos.

**SubtractRect** Obtiene la diferencia entre dos rectángulos.

La intersección de dos rectángulos es siempre un rectángulo, hay un caso especial, cuando los rectángulos no tienen ninguna superficie en común. En ese caso, el resultado es un rectángulo vacío.

En el caso de uniones, se define la unión de dos rectángulos como el rectángulo más pequeño que incluye ambos rectángulos.

La función de diferencia es algo más restrictiva con los rectángulos que admite como parámetros, ya que deben cortarse completamente al menos en uno de los ejes.

Veremos el uso de rectángulos sobre la marcha, como los hemos visto hasta ahora, sin haberlos explicado en detalle.

## Ejemplo 24

## Regiones

Las regiones son figuras como rectángulos, elipses o polígonos, o combinaciones de ellos. Y sus aplicaciones son análogas a las de los rectángulos, aunque como veremos, más completas.

Las regiones se manejan mediante manipuladores, y se puede usar para invalidar partes del área de cliente, [InvalidateRgn](#).

## Funciones para regiones

Veremos ahora algunas de las funciones de las que disponemos para trabajar con regiones.

### Crear regiones



Como en el caso de los rectángulos, disponemos de varias funciones para crear regiones:

Función	Utilidad
CreateRectRgn	Crea una región rectangular.
CreateRectRgnIndirect	Crea una región rectangular a partir de una estructura RECT.
CreateRoundRectRgn	Crear una región rectángulas con las esquinas redondeadas.
CreateEllipticRgn	Crea una región elíptica.
CreateEllipticRgnIndirect	Crea una región elíptica a partir de una estructura RECT que define el rectángulo que inscribe a la elipse.
CreatePolygonRgn	Crea una región poligonal.
CreatePolyPolygonRgn	Crea una región compuesta por varios polígonos.

Cualquier región se puede seleccionar para un contexto de dispositivo, usando la función [SelectObject](#).

Disponemos de un amplio conjunto de operaciones que se pueden realizar con regiones: combinarlas, compararlas, obtener sus dimensiones, pintarlas, recuadrarlas, etc.

## Combinar regiones

Para combinar regiones se usa la función [CombineRgn](#), esta función permite combinar dos regiones de cinco formas diferentes:

Combinación	Resultado
RGN_AND	La nueva región es la intersección de las dos regiones combinadas.

<b>RGN_COPY</b>	El resultado no tiene en cuenta la segunda región, es siempre una copia de la primera.
<b>RGN_DIFF</b>	La nueva región es la diferencia entre las dos regiones, la primera menos la parte común.
<b>RGN_OR</b>	La nueva región es la suma de las dos regiones combinadas.
<b>RGN_XOR</b>	La nueva región es la suma de las dos regiones combinadas, excluyendo las partes comunes.

## Comparar regiones

Podemos comparar dos regiones usando la función [EqualRgn](#), dos regiones se consideran iguales si tienen el mismo tamaño y forma.

## Rellenar regiones

Para rellenar regiones se usa la función [FillRgn](#), al igual que vimos en el [capítulo 21](#), el modo de relleno de polígonos también afecta a la hora de rellenar regiones. Podemos por lo tanto, usar las funciones [SetPolyFillMode](#) y [GetPolyFillMode](#) para cambiar o consultar el modo de relleno actual.

Otra función parecida es [PaintRgn](#), que usa el pincel actual, en lugar de tener que especificar uno. Los modos de relleno de polígonos afectan del mismo modo que en [FillRgn](#).

La función [InvertRgn](#) invierte los colores presentes en la pantalla para la región especificada. En blanco y negro, invertir significa cambiar los pixels blancos por negros y viceversa. En color, el resultado depende del tipo de dispositivo.

El otro modo de mostrar una región es enmarcarla, para eso se usa la función [FrameRgn](#). Enmarcar una región significa trazar una

línea a su alrededor, para lo que hay que especificar un pincel y la anchura y altura del marco.

## Mover una región

Mediante la función [OffsetRgn](#) podemos desplazar una región en cualquier dirección.

## Comprobar posiciones

Tenemos dos funciones para hacer verificaciones sobre regiones, [PtInRegion](#) verifica si un punto está en el interior de una región o no. Esto nos será útil cuando trabajemos con el ratón, para saber si el cursor pasa por determinadas zonas de la ventana.

La otra función es [RectInRegion](#), que verifica si alguna parte de un rectángulo está dentro de una región determinada.

En próximos capítulos veremos como podemos usar una región o un camino para definir un área de recorte. Fuera de la zona definida como área de recorte no se producirá ninguna salida gráfica. Esto nos permite crear formas elaboradas.

## Destruir regiones

Como otros objetos del GDI, hay que destruir las regiones cuando ya no nos hagan falta, liberando de ese modo los recursos usados. Para destruir una región se usa la función [DeleteObject](#).

## Ejemplo 25

# Capítulo 26 Objetos básicos del GDI: El camino (Path)

Se usan para crear figuras complejas, a base de unir segmentos rectos con curvas y líneas Bézier. Estas figuras también pueden contener zonas rellenas y texto.

Los caminos siempre están asociados a un contexto de dispositivo, pero al contrario que otros objetos del GDI, no existe un camino por defecto.

## Crear un camino

Para crear un camino hay que definir los puntos que lo componen, esto se hace usando funciones de trazado del GDI entre las llamadas a las funciones [BeginPath](#) y [EndPath](#).

Las funciones que se pueden usar dentro de un camino son:

<a href="#">AngleArc</a>	<a href="#">LineTo</a>	<a href="#">Polyline</a>
<a href="#">Arc</a>	<a href="#">MoveToEx</a>	<a href="#">PolylineTo</a>
<a href="#">ArcTo</a>	<a href="#">Pie</a>	<a href="#">PolyPolygon</a>
<a href="#">Chord</a>	<a href="#">PolyBezier</a>	<a href="#">PolyPolyline</a>
<a href="#">CloseFigure</a>	<a href="#">PolyBezierTo</a>	<a href="#">Rectangle</a>
<a href="#">Ellipse</a>	<a href="#">PolyDraw</a>	<a href="#">RoundRect</a>
<a href="#">ExtTextOut</a>	<a href="#">Polygon</a>	<a href="#">TextOut</a>

```
BeginPath(hdc);  
SetBkMode(hdc, TRANSPARENT);  
TextOut(hdc, 10,10, "Con Clase", 9);  
Rectangle(hdc, 0,0,10,10);  
EndPath(hdc);
```

La función [CloseFigure](#) sirve para cerrar figuras irregulares creadas a partir de segmentos rectos y/o curvas.

En cualquier momento, antes de cerrar un camino, podemos eliminarlo usando [AbortPath](#).

## Operaciones con caminos

En el momento de cerrar el camino, llamando a [EndPath](#), se selecciona el camino y se borra el previamente seleccionado para ese DC. A partir de ese momento tenemos varias opciones:

<a href="#">StrokePath</a>	Trazar la línea definida por el camino, usando la pluma actual.
<a href="#">FillPath</a>	Pintar el interior del camino, usando el pincel actual.
<a href="#">StrokeAndFillPath</a>	Ambas cosas.
<a href="#">PathToRegion</a>	Convertir el camino en una región.
<a href="#">FlattenPath</a>	Vectorizar el camino, convertir las curvas en series de segmentos rectos que se aproximen.
<a href="#">GetPath</a>	Recuperar las coordenadas y tipos de los puntos que componen el camino.
<a href="#">SelectClipPath</a>	Convertir el camino en un camino de recorte.

Como en anteriores ocasiones, el proceso de rellenar figuras está sujeto al modo de relleno de polígonos, podemos obtener ese modo llamando a [GetPolyFillMode](#) y cambiarlo usando [SetPolyFillMode](#).

El tema de recortes se trata con detalle en el siguiente capítulo.

## Ejemplo 26

# Capítulo 27 Objetos básicos del GDI: El recorte (Clipping)

El recorte nos permite limitar las salidas del GDI a una determinada zona, definida por una región o por un camino.

La región de recorte es uno de los objetos del GDI que podemos seleccionar en un contexto de dispositivo. Del mismo modo que seleccionamos pinceles, brochas, fuentes, etc.

## Regiones de recorte y el mensaje WM\_PAINT

Algunos contextos de dispositivo tienen una región de recorte por defecto, por ejemplo, si obtenemos un DC mediante la función [BeginPaint](#), el DC tendrá una región de recorte correspondiente a la región invalidada que ha provocado el mensaje [WM\\_PAINT](#).

Ya hemos usado la función [InvalidateRect](#) para forzar la actualización de parte del área de cliente, normalmente usamos un rectángulo nulo, con lo que se actualiza toda la ventana.

En ese caso, la región de recorte del DC será el rectángulo especificado.

También podemos usar la función [InvalidateRgn](#). Tanto en un caso como en el otro, las sucesivas llamadas a estas funciones actualizan la región de recorte, de modo que la función [BeginPaint](#) obtiene la región de recorte resultante.

Otras formas en que esa región se actualiza es cuando parte del área de cliente se oculta porque otras ventanas se superponen. Los rectángulos de esas ventanas componen la región de recorte final, que se recibe al procesar el mensaje [WM\\_PAINT](#).

Si se obtiene un DC mediante [CreateDC](#) o [GetDC](#), no tendremos una región de recorte por defecto.

## Funciones relacionadas con el recorte

Existen algunas funciones que se pueden aplicar a la región de recorte asociada a un DC sin necesidad de tener un manipulador de región, directamente sobre el DC:

<a href="#">PtVisible</a>	Nos sirve para determinar si un punto está dentro de los límites de la región asociada a un DC.
<a href="#">RectVisible</a>	Nos permite determinar si una parte de un rectángulo será visible en en DC.
<a href="#">OffsetClipRgn</a>	Para mover la región de recorte en el desplazamiento que queramos.
<a href="#">ExcludeClipRect</a>	Elimina un área rectangular de la región de recorte actual.
<a href="#">IntersectClipRect</a>	Limita la región de recorte actual a su intersección con un rectángulo dado.

## Seleccionar regiones de recorte

Además de las funciones que hemos visto: [InvalidateRect](#) e [InvalidateRgn](#), podemos crear una región de recorte directamente a partir de una región.

Como vimos en el capítulo 25, existen funciones que se aplican directamente sobre regiones. Podemos aplicar estas funciones a la región de recorte si previamente obtenemos esa región mediante [GetClipRgn](#), y una vez modificada, volvemos a aplicar la región de recorte mediante [SelectClipRgn](#).

Si usamos un manipulador de región **NULL**, crearemos una región de recorte nula.

En el ejemplo de este capítulo, cada vez que se debe dibujar el área de cliente, el contenido es diferente, (se trata de líneas paralelas, cada vez más juntas, y alternativamente verticales y

horizontales, y en colores rojo y azul). De este modo podemos ver claramente la región de recorte, ya que únicamente esa región se trazará cada vez.

Hemos puesto una opción para crear una zona de recorte rectangular, que obtengamos usando la función [InvalidateRect](#), y otra para una zona elíptica, que obtenemos usando [InvalidateRgn](#) con una región que hemos creado mediante [CreateEllipticRgn](#).

Un detalle importante es que la región de recorte se va construyendo a medida que se invalidan las distintas zonas, y sólo cuando el sistema "decide" procesar el mensaje [WM\\_PAINT](#) se actualiza la región de recorte tal como existe en ese momento.

La decisión de procesar el mensaje [WM\\_PAINT](#) depende de la carga de trabajo del sistema, y del modo en que se procesan las distintas órdenes de invalidar, de modo que siempre tiene algo de aleatorio.

También podemos experimentar lo que pasa si partes de la ventana de cliente se tapan por otras ventanas. Veremos que se crea una región con todas las zonas del área de cliente que se han tapado, y que al pasar a primer plano, sólo esas zonas se actualizan.

## **Caminos de recorte**

Además de regiones, podemos usar caminos para definir áreas de recorte. Los caminos nos proporcionan algo más de flexibilidad, ya que podemos usar curvas Bézier para definir sus figuras.

Normalmente usaremos caminos para definir áreas de recorte con el objetivo de crear efectos especiales, como hicimos en el ejemplo del capítulo anterior, creando un camino a partir de un texto, y dibujando puntos aleatoriamente dentro de ese camino.

En el ejemplo de este capítulo usamos un camino de recorte, y también una región para crear efectos con puntos o líneas que se adaptan a ciertas figuras.



Se usa la función [SelectClipPath](#) para crear un área de recorte a partir de un camino, y la función [SelectClipRgn](#) para hacerlo a partir de una región.

## **Ejemplo 27**

# Capítulo 28 Objetos básicos del GDI: Espacios de coordenadas y transformaciones

## Definiciones

Un **sistema de coordenadas** es una representación del espacio plano basado en un sistema Cartesiano. Es decir, dos ejes perpendiculares.

En Windows, el espacio es limitado, es decir, el valor máximo de las coordenadas está acotado.

En Windows se usan cuatro sistemas de coordenadas:

- El del mundo. Se refiere al espacio de la aplicación. Permite usar  $2^{32}$  unidades en cada eje.
- El de la página. Es lo que normalmente denominamos espacio lógico, es decir, el espacio donde se usan las unidades lógicas, aunque también en el espacio del mundo trabajaremos con unidades lógicas. Permite usar  $2^{32}$  unidades en cada eje.
- El del dispositivo. En este espacio podemos trabajar con unidades como pulgadas o milímetros. Permite usar  $2^{27}$  unidades en cada eje.
- El del dispositivo físico. Generalmente se refiere al área de cliente, aunque también puede tratarse de la página de la impresora o del ploter. Su tamaño es variable, y depende de cada dispositivo.

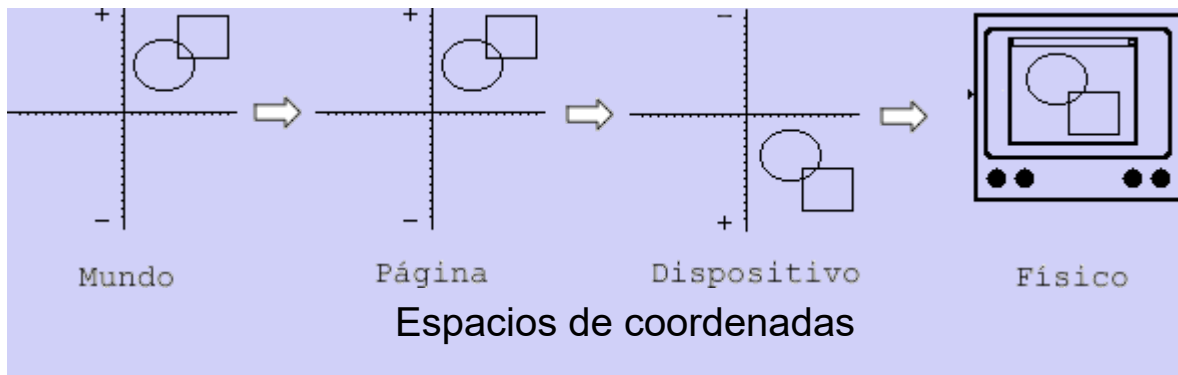
Las **transformaciones** son algoritmos que se usan para hacer conversiones entre el espacio de coordenadas del mundo y el de la

página, de modo que nos es posible realizar cambios de escala, rotaciones, traslaciones, deformaciones o reflexiones.

Esta característica se introdujo en el API de Win32, y no está disponible para versiones de Windows 95 y anteriores.

En Windows usaremos también otro tipos de proyecciones de coordenadas: el mapeo. El **mapeo** consiste en una transformación, aunque algo más limitada, ya que sólo permite escalar, trasladar y cambiar la orientación de los ejes.

El mapeo se aplica entre el espacio de página y el de dispositivo, y existe desde la creación de Windows. Nos permite trabajar con unidades físicas, como pulgadas o milímetros, y usar la orientación del espacio de la pantalla o el papel, en el que las *y* crecen hacia abajo, o la orientación matemática tradicional, en el que las *y* crecen hacia arriba.



Espacios de coordenadas

## Transformaciones

Ya hemos mencionado que las transformaciones y el espacio de coordenadas del mundo son elementos nuevos dentro del API de Win32. De modo que estas características no funcionan con versiones previas a Windows NT, ni siquiera con Windows 95.

Las transformaciones usan un par de fórmulas sencillas para realizar el cambio de coordenadas del espacio del mundo al espacio de página. Si  $(x,y)$  son las coordenadas en el espacio del mundo, y

(x',y') son las coordenadas en el espacio de página, las fórmulas para obtener estas coordenadas son:

$$\begin{aligned}x' &= x * eM_{11} + y * eM_{21} + eD_x \\y' &= x * eM_{12} + y * eM_{22} + eD_y\end{aligned}$$

Estas fórmulas se pueden expresar mediante cálculo matricial:

$$\begin{vmatrix} x' & y' & 1 \end{vmatrix} = \begin{vmatrix} x & y & 1 \end{vmatrix} \cdot \begin{vmatrix} eM_{11} & eM_{12} & 0 \\ eM_{21} & eM_{22} & 0 \\ eD_x & eD_y & 1 \end{vmatrix}$$

La tercera matriz es la **matriz de transformación**. Esta matriz se maneja en el API mediante una estructura [XFORM](#), y como los valores de la tercera columna son conocidos, no se guardan.

Si no modificamos la matriz de transformación, el sistema usa una **matriz identidad**, en la que todos los elementos son nulos, excepto la diagonal principal:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Si aplicamos esta matriz de transformación, no se produce ninguna transformación:

$$\begin{aligned}x' &= x * eM_{11} + y * eM_{21} + eD_x = x*1 + y*0 + 0 = x \\y' &= x * eM_{12} + y * eM_{22} + eD_y = x*0 + y*1 + 0 = y\end{aligned}$$

## Traslaciones

Si analizamos cada término, veremos que  $eD_x$  y  $eD_y$  nos permiten hacer traslaciones, es decir, mover puntos en cualquier

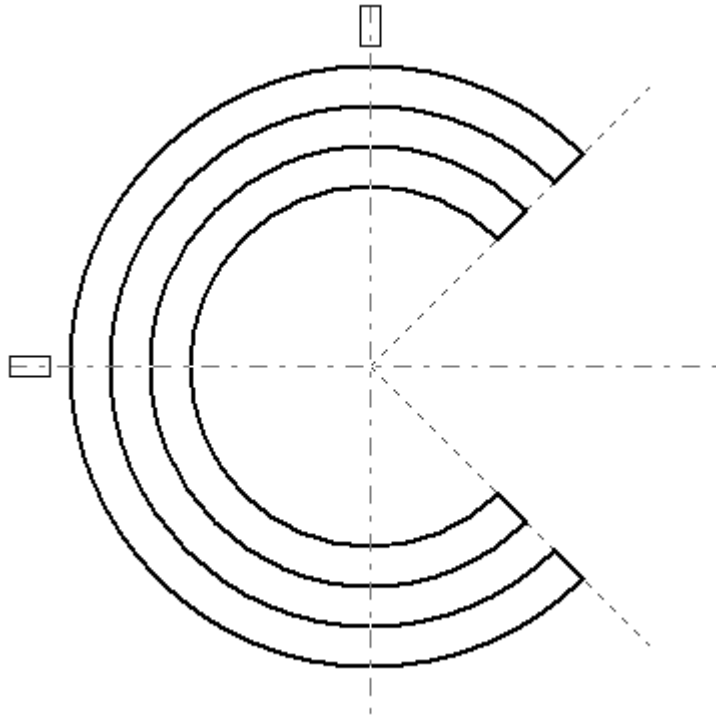


Figura original

dirección.  $(eD_x, eD_y)$  es, sencillamente, un desplazamiento.

## Cambio de escala

Si los términos  $eM_{21}$  y  $eM_{12}$  son nulos, podemos ver que  $eM_{11}$  y  $eM_{22}$  realizan un cambio de escala.

$$\begin{vmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

## Rotaciones

Las rotaciones implican algunos cálculos trigonométricos sencillos, para rotar la figura un ángulo  $\alpha$ , se aplica la siguiente fórmula:

$$\begin{aligned} x' &= x * \cos(\alpha) + y * \sin(\alpha) \\ y' &= x * (-\sin(\alpha)) + y * \cos(\alpha) \end{aligned}$$

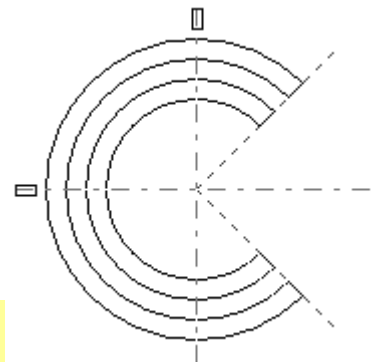


Figura escalada

De donde se deduce que:

- $eM_{11}$  es el coseno de  $\alpha$
- $eM_{12}$  es el seno de  $\alpha$
- $eM_{21}$  es menos el seno de  $\alpha$
- $eM_{22}$  es el coseno de  $\alpha$

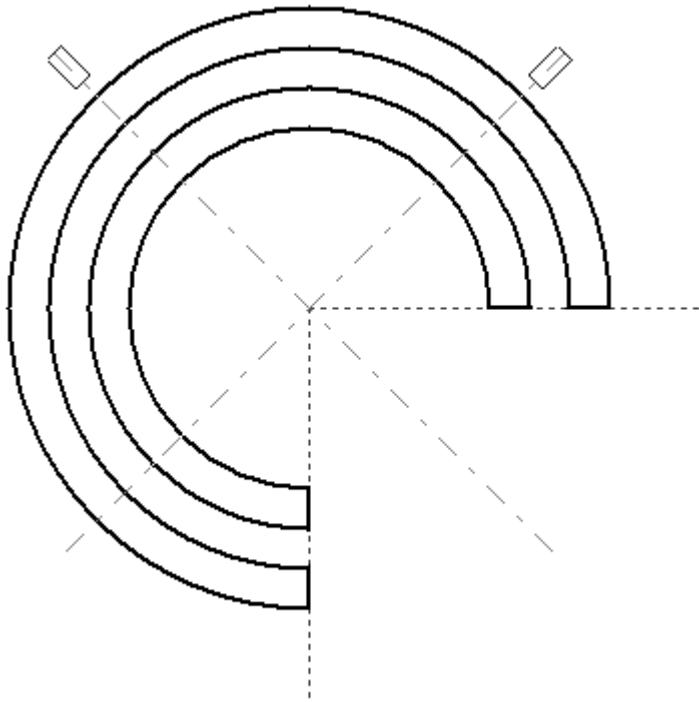


Figura rotada

*math.h*, y se llaman *sin* y *cos*.

#### Nota:

Cuidado con los cálculos, hay que tener en cuenta que en C y C++ se usan ángulos expresados en radianes,  $360^\circ$  son  $2\pi$  radianes ( $90^\circ$  son  $\pi/2$  radianes, y  $45^\circ$  son  $\pi/4$  radianes). Además, las funciones trigonométricas están declaradas en el fichero de cabecera

## Cambio de ejes

Podemos inclinar los ejes, de forma que la transformación no sea ortogonal (es decir, que los ejes no sean perpendiculares). Bastará considerar los factores  $eM_{12}$  y  $eM_{21}$  como constantes de proporcionalidad, horizontal y vertical, respectivamente.

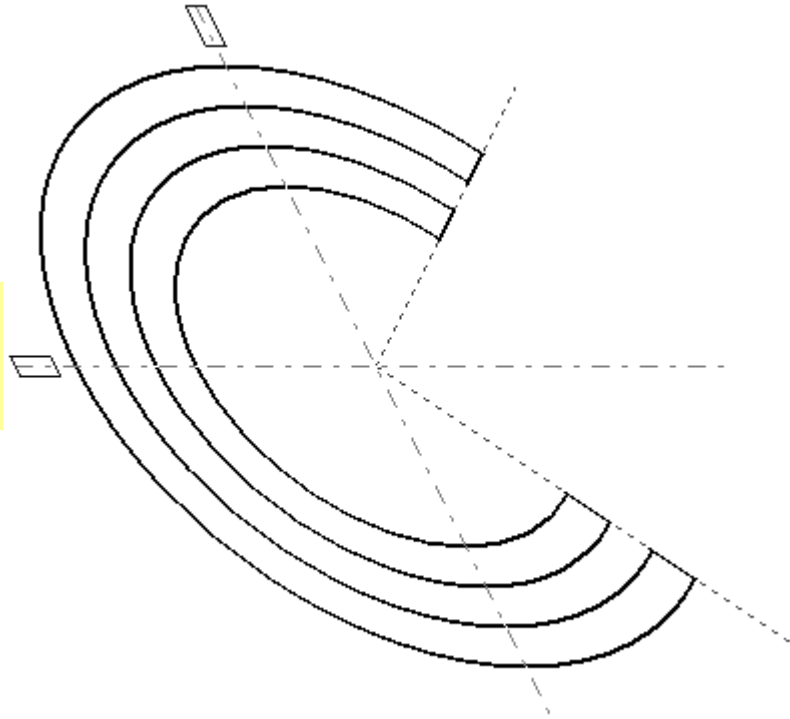
$$\begin{aligned} x' &= x + y * eM_{21} \\ y' &= x * eM_{12} + y \end{aligned}$$

## Reflexiones

Podemos considerar una reflexión como un caso particular de cambio de escala. Si cambiamos el signo de  $eM_{11}$  obtendremos una

reflexión en el eje y,  
si cambiamos el  
signo de  $eM_{22}$   
obtendremos una  
reflexión en el eje x.

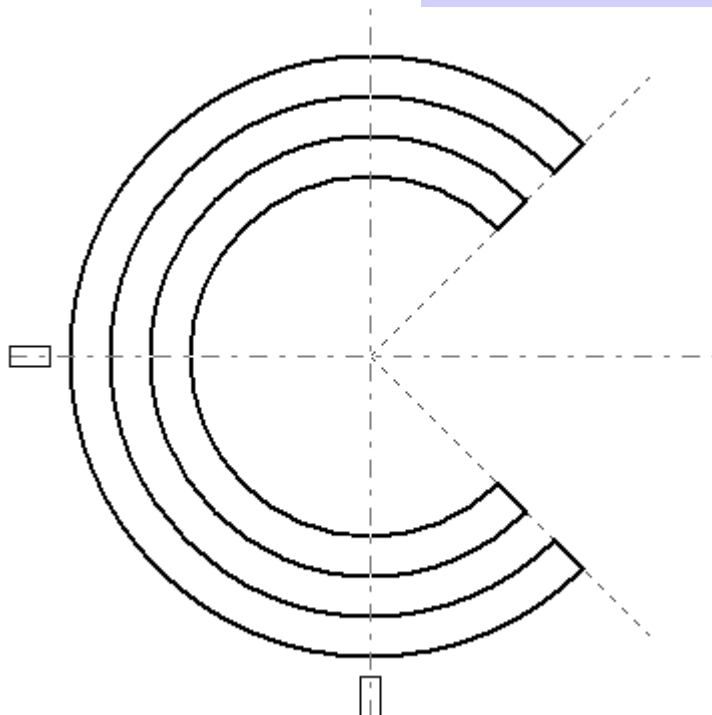
$$\begin{aligned}x' &= x * eM_{11} \\ y' &= y * eM_{22}\end{aligned}$$



## Aplicar transformaciones

En el API32 hay

Cambio de eje



Reflexión

dos modos gráficos diferentes. El compatible y el avanzado.

El compatible es el modo original de Windows 3.1 y Windows 95, y el único que existía en esos sistemas.

El modo avanzado sólo existe en el API32, para Windows NT y sistemas posteriores, como ME, 2000 y XP. Sólo en este modo es posible usar transformaciones, de modo que antes de

aplicar cualquier transformación, será necesario activar el modo avanzado.

Para cambiar el modo gráfico se usa la función [SetGraphicsMode](#), el modo gráfico se aplica a un DC, así que necesitamos un manipulador de DC. Disponemos de dos constantes para activar cada uno de los modos: [GM\\_COMPATIBLE](#) y [GM\\_ADVANCED](#).

Una vez activado el modo avanzado, ya podemos usar transformaciones. Si no aplicamos ninguna, por defecto se usa la transformación identidad. Y si queremos activar el modo compatible, es imprescindible que la transformación actual sea la de identidad, de otro modo la función [SetGraphicsMode](#) fallará.

Para aplicar una transformación usaremos la función [SetWorldTransform](#), que requiere un manipulador de DC y un puntero a una estructura [XFORM](#) con la transformación a aplicar.

```
XFORM xform = {1, 0, 0, 1, 0, 0}; // Sin transformar
(matriz identidad)
...
case WM_PAINT:
    hdc = BeginPaint(hwnd, &ps);
    SetGraphicsMode(hdc, GM_ADVANCED);
    SetWorldTransform(hdc, &xform);
    // Funciones de trazado gráfico
    EndPaint(hwnd, &ps);
    break;
```

## Combinar transformaciones

No tenemos que limitarnos a hacer transformaciones simples, podemos combinarlas para crear transformaciones complejas, de modo que podemos rotar, trasladar, cambiar ejes, escalar y reflejar mediante una única transformación.

Para ello disponemos de dos opciones diferentes:



**Combinar dos transformaciones** mediante la función [CombineTransform](#). Esta función obtiene una transformación a partir de otras dos, el resultado de aplicar la transformación obtenida equivale a aplicar las dos transformaciones, una a continuación de la otra.

**Modificar la transformación del mundo actual** mediante la función [ModifyWorldTransform](#). Mediante esta función será posible combinar la transformación actual con otra, o bien, asignar la matriz de transformación identidad (si se usa el valor [MWT\\_IDENTITY](#) como tercer parámetro. Este tercer parámetro admite otros dos valores: [MWT\\_LEFTMULTIPLY](#) y [MWT\\_RIGHTMULTIPLY](#), que permite multiplicar la transformación indicada por la izquierda o por la derecha. El resultado puede ser diferente, ya que la multiplicación de matrices no posee la propiedad conmutativa.

Si necesitamos obtener la matriz de transformación actual, podemos hacer uso de la función [GetWorldTransform](#).

## Cambios de escala y plumas

Cuando se usan plumas cosméticas para trazar figuras, las únicas que hemos usado hasta ahora, el grosor del trazo se expresa en unidades lógicas, es decir, si duplicamos la escala, el grosor de las líneas se duplica. Esto es así salvo que indiquemos un grosor 0. En ese caso, las líneas siempre son de un pixel de ancho, y por lo tanto, el resultado es que parece que las líneas son más finas cuanto más grande sea la escala. Este efecto puede ser útil cuando trazamos ejes o líneas de ayuda.

## Ejemplo 28

## Ventanas y viewports

La ventana define en el espacio de coordenadas de la página, mediante dos parámetros: la extensión y el origen.

El *viewport* define el espacio de coordenadas del dispositivo, mediante los mismos parámetros que la ventana: la extensión, y el origen.

En el caso del viewport, al definir el espacio del dispositivo, los valores se expresan en coordenadas de dispositivo, es decir, en pixels.

Los puntos en el espacio de página se expresan en coordenadas lógicas, y por lo tanto, también se usan valores lógicos en la ventana. Tanto la extensión como el origen de la ventana se expresa en valores lógicos.

No debemos confundir el espacio de página con la ventana física de la aplicación, aunque existe cierta analogía, la ventana de la que hablamos ahora es un concepto de espacio gráfico, no siempre ligado a la ventana que se muestra en el monitor, es más bien, una ventana que nos permite ver parte del espacio gráfico total.

## Extensiones

En el caso de la ventana la extensión se ajusta por la función [SetWindowExtEx](#). También existe una función para obtener ese parámetro: [GetWindowExtEx](#).

Para el *viewport* también existe una pareja de funciones para asignar y leer la extensión: [SetViewportExtEx](#) y [GetViewportExtEx](#).

## Orígenes

Tanto en el caso de la ventana como en el del viewport, podemos definir un origen de coordenadas, en el caso de la ventana se usa la función [SetWindowOrgEx](#) y en el caso del viewport, la función [SetViewportOrgEx](#). Por supuesto, existen las funciones simétricas, para leer esas coordenadas: [GetWindowOrgEx](#) y [GetViewportOrgEx](#).

# Mapeos

Podríamos titular este apartado como transformaciones del espacio de página al de dispositivo.

Para realizar estas transformaciones se usan los valores de extensión y origen de la ventana y del viewport. Los puntos situados en la ventana se proyectan (o se mapean) al espacio del viewport. Las fórmulas para hacer esas proyecciones son simples:

$$\begin{aligned}D_x &= ((L_x - WO_x) * VE_x / WE_x) + VO_x \\D_y &= ((L_y - WO_y) * VE_y / WE_y) + VO_y\end{aligned}$$

Donde:

- $(D_x, D_y)$  son las coordenadas del punto en unidades de dispositivo.
- $(L_x, L_y)$  las coordenadas del puntir en unidades lógicas (unidades del espacio de página).
- $(WO_x, WO_y)$  las coordenadas del origen de la ventana.
- $(VO_x, VO_y)$  las coordenadas del origen del viewport.
- $(WE_x, WE_y)$  es la extensión de la ventana.
- $(VE_x, VE_y)$  la extensión del viewport.

Básicamente, estas fórmulas definen un cambio de escala, y una traslación.

Existen dos funciones para realizar estos cálculos, así como sus inversos. Podemos, de este modo, pasar las coordenadas de un punto de un espacio al otro. Para pasar un punto en coordenadas lógicas (de página) a coordenadas de dispositivo, se usa la función [LPtoDP](#). Para pasar de coordenadas de dispositivo a coordenadas lógicas, se usar la función [DPtoLP](#).

## Modos de mapeo predefinidos

En Windows existen ocho posibles modos de mapeo, cada uno con sus características propias:

- Isotrópico: el mapeo entre el espacio de página y el del dispositivo se define por la aplicación, cambiando las extensiones y orígenes de la ventana y del *viewport*. Las medidas en ambos ejes son iguales, pero la orientación se puede definir por la aplicación.
- No isotrópico: igual que el anterior, pero las medidas en el eje x no tienen por qué ser iguales que en el eje y.
- Inglés alto: cada unidad del espacio de página se mapea a 0.001 pulgadas en el espacio de dispositivo. Las x crecen hacia la derecha, las y hacia arriba.
- Inglés bajo: cada unidad del espacio de página se mapea a 0.01 pulgadas en el espacio de dispositivo. Las x crecen hacia la derecha, las y hacia arriba.
- Métrico alto: cada unidad del espacio de página se mapea a 0.01 milímetros en el espacio de dispositivo. Las x crecen hacia la derecha, las y hacia arriba.
- Métrico bajo: cada unidad del espacio de página se mapea a 0.1 milímetros en el espacio de dispositivo. Las x crecen hacia la derecha, las y hacia arriba.
- Texto: cada unidad del espacio de página se mapea a un pixel, es decir, no se hace ningún cambio de escala.
- Twips: cada unidad del espacio de página se mapea a 1/20 de punto de impresora (un twip), que equivale a 1/1440 de pulgada. Las x crecen hacia la derecha, las y hacia arriba.

Para activar un modo de mapeo se usa la función [SetMapMode](#). Esta función precisa dos parámetros: un manipulador de DC, y el identificador del modo de mapeo ([MM\\_ISOTROPIC](#), [MM\\_ANISOTROPIC](#), [MM\\_HIENGLISH](#), [MM\\_LOENGLISH](#), [MM\\_HIMETRIC](#), [MM\\_LOMETRIC](#), [MM\\_TEXT](#) y [MM\\_TWIPS](#), respectivamente). Para averiguar el modo actual [GetMapMode](#).

En el caso de los seis últimos modos (todos menos el isotrópico y el no isotrópico), la extensión del viewport no puede alterarse, ya

que se ajusta automáticamente, en función de la extensión de la ventana.

Cuando se usan los modos isotrópico o no isotrópico, es necesario ajustar la extensión y origen del *viewport*, ya que en estos modos, el cambio de escala se define por la aplicación. En el caso del modo isotrópico es importante ajustar la extensión de la ventana antes de hacerlo con el *viewport*.

## Modo por defecto

El modo por defecto es el de texto, que además es el único que es dependiente del dispositivo.

Este modo es útil si la salida sólo se va a visualizar en pantalla, pero no resultará muy conveniente si tenemos que asegurar que el tamaño de nuestros gráficos es real (por ejemplo si dibujamos un cuadrado de 10 centímetros de lado), y mucho menos si tenemos que realizar salidas a impresora, si trazamos un cuadrado de 100 pixels en pantalla, el tamaño en impresora dependerá mucho de los puntos por pulgada de resolución que tenga la impresora.

En cada caso será interesante escoger un modo independiente de dispositivo, ya sea en pulgadas, milímetros, o puntos de impresora.

## Transformaciones definidas por el usuario

Hay dos modos de mapeo que implican que la transformación se define por el usuario: el isotrópico y el no isotrópico. Son modos muy similares, pero el isotrópico asegura que las unidades lógicas son del mismo tamaño en el eje x y el en eje y. El modo no isotrópico permite usar unidades diferentes en cada eje.

Por ejemplo, para obtener una resolución de 1/3 milímetro, podemos usar el modo isotrópico de esta manera:

```
SetMapMode(hDC, MM_ISOTROPIC);  
SetWindowExtEx(hDC, 3*GetDeviceCaps(hDC, HORZSIZE),  
               3*GetDeviceCaps(hDC, VERTSIZE), NULL);  
SetViewportExtEx(hDC, GetDeviceCaps(hDC, HORZRES),  
                 GetDeviceCaps(hDC, VERTRES), NULL);
```

Invocamos la función [GetDeviceCaps](#) para obtener las dimensiones del dispositivo en milímetros ([HORZSIZE](#),[VERTSIZE](#)), y el pixels ([HORZRES](#),[VERTRES](#)). Hacemos que la extensión x e y de la ventana sea el triple del tamaño del dispositivo en milímetros, y que la extensión del *viewport* sea el tamaño del dispositivo en pixels. Esto hace que cada unidad de página (unidad lógica) se convierta en 1/3 milímetro.

## Modos gráficos y sentido de los arcos

Hay que tener en cuenta que el sentido en que se trazan los arcos es importante, y que el resultado de la salida gráfica dependerá del modo gráfico y del modo de mapeo seleccionados, y en el caso de los modos isotrópico y no isotrópico, de los signos de los valores de las extensiones de ventana y *viewport*.

En el modo gráfico avanzado, los arcos siempre se trazan en el sentido de las agujas del reloj en el espacio lógico. Esto independiza la salida del modo de mapeo elegido. En el caso del modo gráfico compatible se usa el sentido de trazado actual de los arcos para trazarlos en el espacio del dispositivo, es decir, una vez aplicado el mapeo. De modo que si cambiamos el modo de mapeo, el aspecto de los arcos puede cambiar.

Recordemos que podemos cambiar el sentido de trazado de los arcos en el espacio de dispositivo, usando la función [SetArcDirection](#).

**Nota:**

Siempre que podamos, usaremos el modo gráfico avanzado, ya que no sólo nos permite usar transformaciones, sino que además simplifica el trazado de arcos, al no tener que preocuparse del modo de mapeo que se use en cada caso.

## Otras funciones

Hay algunas funciones más relacionadas con espacios de coordenadas:

<code>ClientToScreen</code>	Convierte coordenadas de cliente a coordenadas de pantalla.
<code>ScreenToClient</code>	Convierte coordenadas de pantalla a coordenadas de cliente.
<code>GetCurrentPositionEx</code>	Recupera la posición actual del cursor gráfico en coordenadas lógicas.
<code>OffsetWindowOrgEx</code>	Desplaza el origen de la ventana en las cantidades especificadas.
<code>OffsetViewportOrgEx</code>	Desplaza el origen del <i>viewport</i> en las cantidades especificadas.
<code>ScaleWindowExtEx</code>	Modifica la extensión de la ventana en el factor especificado.
<code>ScaleViewportExtEx</code>	Modifica la extensión del <i>viewport</i> en el factor especificado.

## Ejemplo 29

# Capítulo 29 Objetos básicos del GDI: Plumas geométricas

En el capítulo 18 vimos cómo crear y usar plumas cosméticas, y hablamos un poco de plumas geométricas, aunque sin entrar en detalles. En este capítulo veremos cómo crear, usar y destruir plumas geométricas, así como sus características y propiedades.

## Atributos de las plumas geométricas

Las plumas geométricas tienen siete atributos: anchura, estilo, color, patrón, rayado, estilo de extremos y estilo de uniones. Recordemos que las plumas cosméticas sólo tenían los tres primeros.

Los patrones y el rayado son atributos que hasta ahora sólo hemos asociado a pinceles, pero también los poseen las plumas geométricas.

Esto hace que las plumas geométricas sean más difíciles de crear, y más lentas a la hora de trazar líneas, pero son mucho más potentes y versátiles que las cosméticas.

Para crear plumas geométricas usaremos la función [ExtCreatePen](#).

Veamos ahora algunos de esos atributos en detalle, y qué opciones existen.

### Anchura

Vimos que en el caso de las plumas cosméticas, el grosor se expresaba en unidades de dispositivo, es decir, en pixels. En las geométricas se expresa en unidades lógicas, es decir, su anchura



se ve afectada por las transformaciones del sistema de coordenadas.

Aunque solemos decir que las funciones [CreatePen](#) y [CreatePenIndirect](#) crean plumas cosméticas, esto es falso. Windows sólo crea plumas cosméticas de un pixel (cuando indicamos un grosor de cero en estas funciones), de modo que si creamos plumas más anchas usando estas funciones, en realidad estaremos creando plumas geométricas. Esta limitación de Windows no tiene por qué respetarse en futuras versiones, de modo que estas funciones, podrían crear plumas cosméticas de más de un pixel en el futuro, así que seguiremos diciendo que las plumas creadas usando estas dos funciones son cosméticas, aunque el grosor sea distinto de cero. Pero debemos tener en cuenta que estas plumas se verán afectadas por las transformaciones del espacio del mundo, o por el mapeo.

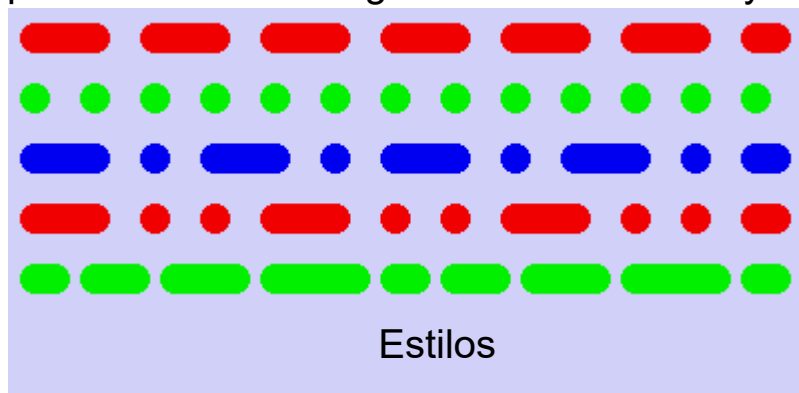
## Estilo de línea

El estilo de línea define el tipo de trazos de las líneas. Recordemos los posibles estilos de línea:

Estilo	Descripción
Sólido	Las líneas serán continuas y sólidas.
Trazos	Líneas de trazos.
Puntos	Líneas de puntos.
Trazo y punto	Líneas alternan puntos y trazos.
Trazo, punto, punto	Líneas alternan líneas y dobles puntos.
Nulo	Las líneas son invisibles.
Dentro de marco	Las líneas serán sólidas. Sólo en el caso de plumas geométricas, y cuando se usan con funciones que requieran un rectángulo que sirva como límite, las dimensiones de la

figura se reducirán para que se ajusten por completo al interior del rectángulo, teniendo en cuenta el grosor de la pluma.

En el caso de las plumas geométricas, los estilos no sólidos no están limitados a un pixel, como en el caso de las plumas cosméticas. Además, existe otro estilo posible, en el que el usuario puede definir las longitudes de los trazos y las separaciones.



## Color

Especifica el color de la pluma. Se puede usar una estructura `COLORREF` para

indicar el color.

## Patrón

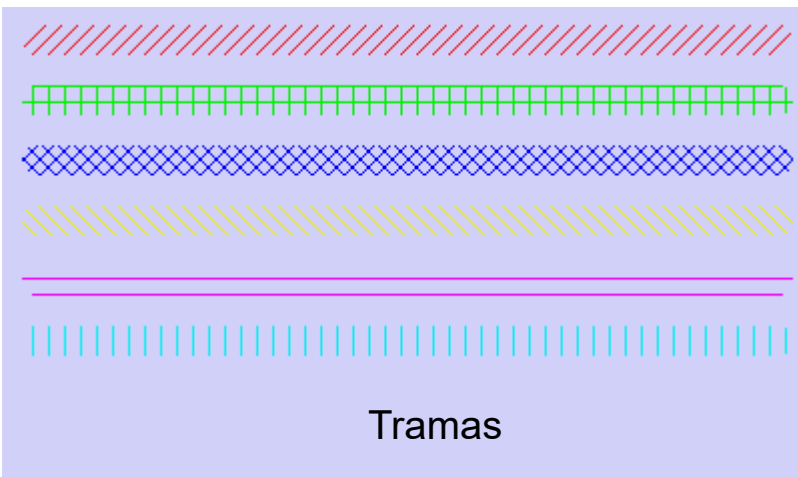
Especifica un patrón de mapa de bits que se repite en el trazo. Uno de los patrones es el de rayado, que puede ser cualquiera de los seis predefinidos. Pero también se pueden usar patrones creados a partir de cualquier mapa de bits de 8x8 pixels, patrones vacíos o sólidos.

## Rayado

El rayado es un tipo particular de patrón. Normalmente nos referiremos a los rayados predefinidos, que son seis, como vimos en el capítulo de plumas:

Valor	Significado
Diagonal descendente	Trama de líneas diagonales a 45° descendentes de izquierda a derecha.
Cruz	Trama de líneas horizontales y verticales.

Cruz diagonal	Trama de líneas diagonales a 45° cruzadas.
Diagonal ascendente	Trama de líneas diagonales a 45° ascendentes de izquierda a derecha.
Horizontal	Trama de líneas horizontales.
Vertical	Trama de líneas verticales.



## Estilo de final (tapón)

Los extremos de las líneas pueden ser de diferentes tipos. Para las plumas geométricas tenemos tres

posibles valores:

Valor	Significado
Redondeado	Las líneas terminan con un semicírculo.
Cuadrado	Las líneas terminan con medio cuadrado.
Plano	Las líneas terminan de forma abrupta.

Por ejemplo, estas líneas fueron trazadas con estos tres estilos de final.



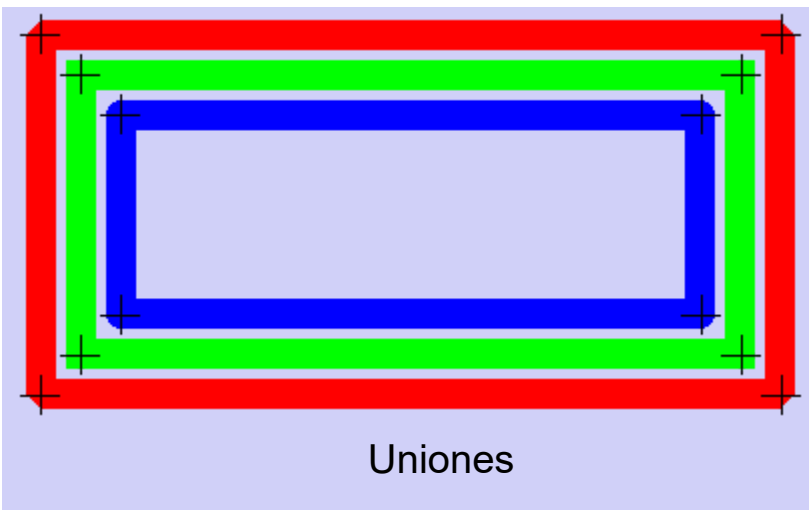
## Estilo de unión

Para las uniones de distintas líneas de una figura cerrada

también disponemos de tres estilos. Estos estilos no se aplican a líneas unidas que no formen parte de una línea poligonal o de una figura cerrada:

Valor	Significado
Redondeado	Las esquinas se redondean.
Picudas (miter)	Las esquinas se dejan sin recortar ni redondear.
Biseladas (Bevel)	Las puntas de las esquinas se cortan a bisel.

Por ejemplo, estas figuras fueron trazadas con estos tres estilos de final. La azul con el estilo redondeado, la verde picuda y la roja viselada:



## Crear una pluma geométrica

Para crear plumas geométricas se usa la función [ExtCreatePen](#):

```
HPEN pluma;
LOGBRUSH lb = {BS_SOLID, RGB(240, 0, 0), 0};

pluma = ExtCreatePen(
    PS_GEOMETRIC | PS_DASH | PS_ENDCAP_ROUND |
    PS_JOIN_ROUND,
    15, &lb, 0, NULL);
```

Como estas plumas pueden usar atributos de pincel para el trazado, necesitamos suministrar una estructura [LOGBRUSH](#) para definir el estilo y color de la pluma.

## Seleccionar una pluma geométrica

No hay diferencia, a la hora de usarlas, entre las plumas geométricas y las cosméticas. En ambos casos, y como sucede con el resto de los objetos del GDI, se usa la función [SelectObject](#):

```
SelectObject(hdc, pluma);  
MoveToEx(hdc, 30, 30, NULL);  
LineTo(hdc, 400, 30);
```

## Destruir una pluma geométrica

Y, por supuesto, cuando ya no necesitemos una pluma geométrica, deberemos destruirla y liberar los recursos que consume, usando la función [DeleteObject](#):

```
DeleteObject(pluma);
```

## Ejemplo 30

# Capítulo 30 Objetos básicos de usuario: El Caret

Los carets son las marcas intermitentes que nos indican dónde se insertará el texto o los graficos cuando un usuario los introduzca. Normalmente son pequeñas líneas verticales u horizontales, aunque pueden ser rectángulos de distintos tonos o incluso mapas de bits.

Ya sabemos que sólo una ventana puede tener el foco en un determinado momento, de modo que sólo puede mostrarse un caret en un momento determinado, precisamente en la ventana que tiene el foco.

Veremos en este capítulo como crear, modificar, ocultar o mostrar carets.

## Recibir y perder el foco

El caret es un recurso compartido, es decir, sólo existe un caret en todo el sistema. Esto implica que debemos tener en cuenta algunos aspectos importantes. Por ejemplo, si nuestra aplicación usa un caret, debe crearlo cada vez que recibe el foco y destruirlo cuando lo pierde, de modo que la nueva aplicación que recibe el foco pueda crear el suyo.

Es importante por lo tanto, saber cuándo nuestra aplicación recibe o pierde el foco, para ello disponemos de dos mensajes muy útiles: [WM\\_SETFOCUS](#) se recibe justo cuando la aplicación recibe el foco, y [WM\\_KILLFOCUS](#) se recibe justo antes de perderlo.

Las aplicaciones que trabajen con carets deben procesar estos dos mensajes. Cuando se recibe [WM\\_SETFOCUS](#) se crea y muestra el caret y cuando se recibe [WM\\_KILLFOCUS](#) se destruye.

## Crear y destruir carets

Para crear un caret usaremos la función [CreateCaret](#), que puede crear cualquiera de los carets posibles: verticales, horizontales, sólidos, grises y de mapas de bits.

Esta función precisa de cuatro parámetros. El primero es el manipulador de ventana para la que creamos el caret. El segundo es un manipulador de mapa de bits, si es **NULL** se creará un caret sólido, si es 1 el caret será gris (trama de puntos), o puede ser un mapa de bits que se usará como figura del caret. El tercer parámetro es la anchura, en unidades lógicas, y el cuarto la altura.

Hay que tener en cuenta que las transformaciones y mapeos afectan al tamaño del caret.

Por ejemplo, para crear un caret vertical:

```
case WM_SETFOCUS:
    CreateCaret(hwnd, (HBITMAP)NULL, 0, 20);
```

Por ejemplo, para crear un caret a partir de un mapa de bits:

```
case WM_SETFOCUS:
    CreateCaret(hwnd, hBitmap, 0, 0);
```

Para destruirlo, cuando perdemos el foco usaremos la función [DestroyCaret](#):

```
case WM_SETFOCUS:
    DestroyCaret();
    break;
```

## Mostrar y ocultar carets

Crear un caret no lo muestra automáticamente, es necesario mostrarlo explícitamente mediante la función [ShowCaret](#). Esto completa la respuesta al mensaje de activación del foco:

```
case WM_SETFOCUS:
    CreateCaret(hwnd, (HBITMAP) NULL, 0, 20);
    ShowCaret(hwnd);
    break;
```

Por otra parte, a veces es necesario ocultar el caret, en particular, cuando se actualiza la pantalla, de modo que no interfiera con el contenido de la ventana. Para ocultar el caret se usa la función [HideCaret](#).

## Procesar mensajes WM\_PAINT

Si no tenemos la precaución de ocultar el caret durante la actualización de la ventana, éste puede interferir con el nuevo contenido, de modo que se corrompa parte del contenido. Cuando nuestra aplicación trabaje con carets y procesemos el mensaje [WM\\_PAINT](#), debemos ocultar el caret mientras duran la operaciones de actualización:

```
case WM_PAINT:
    HideCaret(hwnd);
    hdc = BeginPaint(hwnd, &ps);
    ActualizarPantalla(hdc);
    EndPaint(hwnd, &ps);
    ShowCaret(hwnd);
    break;
```

## Cambiar posición de un caret



Otro parámetro importante con respecto a los carets es su posición en pantalla, ya que se usa para indicar el punto de inserción de texto o de gráficos, normalmente la posición se modificará con mucha frecuencia.

Para cambiar la posición del caret se usa la función [SetCaretPos](#), si necesitamos obtener la posición actual, podemos usar [GetCaretPos](#). La posición del caret se puede modificar aunque el caret esté oculto.

```
SetCaretPos(120,120);
```

## Cambiar velocidad de parpadeo de un caret

El caret tiene otra propiedad importante: el parpadeo. La velocidad del parpadeo se puede modificar usando el Panel de Control, pero el API también proporciona una función para modificar esa velocidad, aunque es nuestra responsabilidad que ese cambio sea a petición del usuario, ya que al tratarse de un recurso compartido, todas las aplicaciones pueden verse afectadas por este cambio.

Para asignar un nuevo valor al parpadeo se usa la función [SetCaretBlinkTime](#) y para obtener la velocidad actual, [GetCaretBlinkTime](#)

```
ActualBlink = GetCaretBlinkTime();  
SetCaretBlinkTime(50);
```

## Ejemplo 31

# Capítulo 31 Objetos básicos del usuario: El icono

Le toca el turno a otro recurso muy familiar para el usuario de Windows: el icono.

Un icono es un pequeño mapa de bits, combinado con una máscara que permite que parte de él sea transparente. El resultado es que las figuras representadas por iconos pueden tener diferentes formas, y no tienen por qué ser rectangulares.

Los usamos como ayuda para representar objetos: directorios, ficheros, aplicaciones, etc.

## Punto activo

Los iconos tienen un punto activo (hot spot), como veremos que sucede también con los cursores. En el caso de los iconos, ese punto suele ser el centro, y se usa para tareas de alineación y espaciado de iconos.

## Tamaños

En Windows se usan iconos en dos entornos, el sistema y el *shell*, y para cada uno de ellos usamos dos tamaños de icono, el grande y el pequeño.

El icono de sistema pequeño se usa en las barras de título de las ventanas. Para averiguar el tamaño de este icono hay que llamar a [GetSystemMetrics](#) con [SM\\_CXSMICON](#) y [SM\\_CYSMICON](#).

El icono de sistema grande es que se usa normalmente en las aplicaciones, las funciones [DrawIcon](#) y [LoadIcon](#) usan por defecto,

estos iconos. Para averiguar el tamaño de este icono se puede llamar a `GetSystemMetrics` con `SM_CXICON` y `SM_CYICON`.

El icono pequeño del shell se usa en el explorador de windows y en los diálogos comunes.

El icono grande del shell se usa en el escritorio.

Cuando creamos iconos a medida en nuestras aplicaciones, deberemos suministrar recursos de icono de los siguientes tamaños:

- 48x48, 256 colores
- 32x32, 16 colores
- 16x16 pixels, 16 colores

## Asociar iconos a una aplicación

Podemos decidir qué iconos usará nuestra aplicación para la barra de título, y qué icono se asocia a la aplicación al registrar la clase de ventana.

Cuando se rellena la estructura `WNDCLASSEX` que se usa para registrar nuestra clase de ventana, el miembro *hIcon* debe ser un icono de 32x32 y el miembro *hIconSm* uno de 16x16. Aunque ya hemos visto en algunos ejemplos que esto es opcional. Si usamos un icono de 32x32 para el *hIconSm* el sistema lo escalará automáticamente a 16x16.

```
wincl.hIcon = LoadIcon (hThisInstance, "tajmahal");  
wincl.hIconSm = LoadIcon (hThisInstance, "lapiz");
```

## Tipos

Existen iconos estándar disponibles para cualquier aplicación, y también es posible modificar las imágenes asociadas a esos iconos estándar, personalizando el escritorio de Windows.

Los iconos estándar son:

Icono	Identificador	Descripción
Aplicación	<code>IDI_APPLICATION</code>	Icono de aplicación por defecto.
Información	<code>IDI_ASTERISK</code>	Asterisco (usado en mensajes de información).
Exclamación	<code>IDI_EXCLAMATION</code>	Signo de exclamación (usado en mensajes de aviso).
Aviso importante	<code>IDI_HAND</code>	Icono de mano extendida (usado en mensajes de aviso importantes).
Interrogación	<code>IDI_QUESTION</code>	Signo de interrogación (usado en mensajes de petición de datos).
Logo	<code>IDI_WINLOGO</code>	Icono de logo de Windows.

Cargar uno de estos iconos es sencillo, basta usar la función [LoadIcon](#), indicando como manipulador de instancia el valor `NULL`, y como identificador de icono, el que queramos cargar:

```
HICON icono = LoadIcon(NULL, IDI_EXCLAMATION);
```

Por otra parte, en nuestras aplicaciones podremos crear nuestros propios iconos, bien a partir de ficheros de recursos, o directamente a partir de datos binarios.

Generalmente usaremos ficheros de recursos, ya que crear iconos durante la ejecución hace que estos sean dependientes del dispositivo, y su aspecto puede ser impredecible.

En el caso de iconos de medidas normales, de 32x32 o de 16x16, se usa la función [LoadIcon](#), en el caso de iconos de 48x48 se debe usar [LoadImage](#):

```
HICON icono1, icono2, icono3;

icono1 = LoadImage(hInstance, "tajmahal", IMAGE_ICON, 0,
0, LR_LOADREALSIZE);
icono2 = LoadIcon(hInstance, "antena");
icono3 = LoadIcon(hInstance, "lapiz");
```

## Iconos en ficheros de recursos

Existen programas de edición de iconos en Internet, como por ejemplo [IconEdit2](#), que es ShareWare. (Hay que registrarse para poder guardar los ficheros de iconos). También podemos optar por usar uno de los muchos iconos de galería que se pueden encontrar en Internet.

Se pueden incluir iconos diseñados por nosotros en el fichero de recursos mediante la sentencia [ICON](#), y obtener un manipulador para ellos usando las funciones [LoadIcon](#) o [LoadImage](#).

```
antena ICON "station.ico"  
lapiz ICON "fayDrafts.ico"  
tajmahal ICON "tajmahal.ico"
```

## Iconos en controles estáticos

Ya hemos usado iconos anteriormente como parte de los controles estáticos en el [capítulo 10](#).

## Mostrar iconos

En futuros capítulos veremos cómo incluirlos en menús o botones, de momento nos conformaremos con mostrarlos en pantalla.

Se puede obtener información mediante [GetIconInfo](#), y mostrar el icono mediante [DrawIconEx](#) o [DrawIcon](#). Si se usa el parámetro [DI\\_COMPAT](#) se mostrará la imagen por defecto, si no, se mostrará la imagen que indiquemos. La función [DrawIconEx](#) es más potente en el sentido de que nos permite mostrar iconos de tamaños diferentes de 32x32, y además nos permite escalarlos.

```
        DrawIconEx(hdc, 10, 10,
                    LoadImage(hInstance, "tajmahal", IMAGE_ICON,
0, 0, LR_LOADREALSIZE),
                    0, 0, 0, NULL, DI_NORMAL);
        DrawIcon(hdc, 60, 10, LoadIcon(hInstance,
"antena"));
        DrawIconEx(hdc, 100, 10, LoadIcon(hInstance,
"lapiz"),
                    16, 16, 0, NULL, DI_NORMAL);
        DrawIcon(hdc, 140, 10, LoadIcon(NULL,
IDI_APPLICATION));
```

## Destrucción de iconos

[DestroyIcon](#) sólo se puede aplicar a iconos creados mediante [CreateIconIndirect](#). No recomiendo usar este tipo de iconos, ya que son dependientes del dispositivo, y pueden producir resultados no deseados en algunos equipos.

## Ejemplo 32

# Capítulo 32 Objetos básicos del usuario: El cursor

El cursor o puntero, indica la posición del ratón en pantalla, y nos permite acceder a los elementos de las ventanas, al tiempo que su aspecto nos da pistas sobre la acción asociada al cursor, o sobre el estado del sistema.

Es un recurso importante y único en el sistema, por lo que hay que compartirlo entre las diferentes aplicaciones. Algunos de los cambios que hagamos en el cursor se deben deshacer cuando el control pase a otras aplicaciones.

## Cursor de clase

Uno de los parámetros que asignamos al registrar una clase de ventana, usando la estructura [WNDCLASS](#) o [WNDCLASSEX](#) y las funciones [RegisterClass](#) o [RegisterClassEx](#) es, precisamente, el cursor de la clase. Windows siempre muestra ese cursor mientras esté dentro del área de cliente de la ventana.

```
    WNDCLASSEX wincl;           /* Estructura de datos para la
clase de ventana */

    /* Estructura de la ventana */
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = "NUESTRA_CLASE";
    wincl.lpfnWndProc = WindowProcedure;      /* Esta
función es invocada por Windows */
    wincl.style = CS_DBLCLKS;                  /* Captura los
doble-clicks */
    wincl.cbSize = sizeof (WNDCLASSEX);

    /* Usar icono y puntero por defecto */
```

```

wincl.hIcon = LoadIcon (hThisInstance, "Icono");
wincl.hIconSm = LoadIcon (hThisInstance, "Icono");
wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
wincl.lpszMenuName = "Menu";
wincl.cbClsExtra = 0; /* Sin
información adicional para la */
wincl.cbWndExtra = 0; /* clase o la
ventana */
/* Usar el color de fondo por defecto para es escritorio
*/
wincl.hbrBackground =GetSysColorBrush(COLOR_BACKGROUND);

/* Registrar la clase de ventana, si falla, salir del
programa */
if(!RegisterClassEx(&wincl)) return 0;

```

Hasta ahora, en todos nuestros ejemplos cargábamos el cursor de la flecha, pero en este capítulo veremos que podemos usar nuestro propio cursor de clase para nuestras ventanas, bastará con asignar otro cursor al miembro *hCursor*. Veremos a continuación qué cursores podemos usar.

## Cursores de recursos

Se pueden incluir cursores diseñados por nosotros o almacenados en ficheros ".cur" en el fichero de recursos mediante la sentencia [CURSOR](#), y obtener un manipulador para ellos usando las funciones [LoadCursor](#) o [LoadImage](#).

```
Flecha3D CURSOR "3dgarro.cur"
```

Para usar estos cursores en nuestra aplicación usaremos la función [LoadCursor](#), indicando la instancia actual y el identificador de recurso:

```
HCURSOR flecha3d = LoadCursor(hInstance, "flecha3D");
```



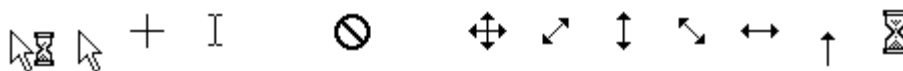
O bien la función [LoadImage](#):

```
HCURSOR flecha3d = LoadImage(hInstance, "flecha3D",  
IMAGE_CURSOR, 0, 0, LR_LOADREALSIZE);
```

## Cursores estándar

Podríamos llamarlos cursores de stock, aunque en realidad no lo son, ya que los cursores estándar se pueden personalizar por el usuario al cambiar las opciones del escritorio.

Existen los siguientes cursores estándar:



### Cursores estándar

Valor	Descripción
<a href="#">IDC_APPSTARTING</a>	Flecha estándar y un pequeño reloj de arena.
<a href="#">IDC_ARROW</a>	Flecha estándar.
<a href="#">IDC_CROSS</a>	Cruz.
<a href="#">IDC_IBEAM</a>	I para texto.
<a href="#">IDC_ICON</a>	Sólo en Windows NT: icono vacío
<a href="#">IDC_NO</a>	Círculo barrado.
<a href="#">IDC_SIZE</a>	Sólo en Windows NT: flecha de cuatro puntas.
<a href="#">IDC_SIZEALL</a>	Igual que <a href="#">IDC_SIZE</a> .
<a href="#">IDC_SIZENESW</a>	Flecha de dos puntas noreste y sudoeste.
<a href="#">IDC_SIZENS</a>	Flecha de dos puntas, norte y sur.
<a href="#">IDC_SIZENWSE</a>	Flecha de dos puntas,

	noroeste y sudeste.
IDC_SIZEWE	Flecha de dos puntas, este y oeste.
IDC_UPARROW	Flecha vertical.
IDC_WAIT	Reloj de arena.

El aspecto gráfico de cada uno depende de la configuración del escritorio de Windows, y se puede modificar usando el Panel de Control. En nuestra aplicación podemos cargar cualquiera de ellos usando la función [LoadCursor](#), indicando **NULL** como manipulador de instancia, y el identificador que queramos.

```
HCURSOR cursor = LoadCursor(NULL, IDC_ARROW);
DrawIconEx(hdc, 10, 120, cursor, 0, 0, 0, NULL,
DI_NORMAL|DI_COMPAT);
```

También podemos cambiar esos cursores mediante la función del API [SetSystemCursor](#). El primer parámetro es un manipulador de cursor, y los valores adecuados para el segundo parámetro son los mismos que en la tabla anterior, pero con el prefijo "OCR\_", en lugar de "IDC\_":

```
HCURSOR flecha3d = LoadCursor(hInstance, "flecha3D");
SetSystemCursor(flecha3d, OCR_NORMAL);
```

Estos cambios son permanentes, en el sentido de que no se restituyen al abandonar la aplicación, y para recuperar los cursores previos hay que usar la misma función o bien el panel de control.

## Similitud entre iconos y cursores

Existe cierta similitud entre cursores e iconos, el formato en el que se guardan es similar, y se puede usar la función [DrawIconEx](#)

para mostrar un icono, tanto como un cursor. Del mismo modo que se puede usar [GetIconInfo](#) para obtener información sobre un cursor.

```
hdc = BeginPaint(hwnd, &ps);
DrawIconEx(hdc, 10, 50, caballo, 0, 0, 0, NULL,
DI_NORMAL|DI_COMPAT);
EndPaint(hwnd, &ps);
```

Además, es posible usar un icono como cursor, aunque con algunas limitaciones, dependiendo del hardware instalado.

```
HICON tajmahal = LoadImage(hInstance, "tajmahal",
IMAGE_ICON, 0, 0, LR_LOADREALSIZE);
SetClassLong(hwnd, GCL_HCURSOR, (LONG)tajmahal);
```

Los cursores pueden ser monocromo, en color, o animados. Aunque algunos sistemas sólo admiten determinados tipos de cursores, monocromo y de determinadas dimensiones, sobre todo antes de windows 95. Por ejemplo, no se pueden usar cursores en color con una pantalla VGA.

## El punto activo (Hot Spot)

La similitud entre iconos y cursores se da también en la propiedad del punto activo. En el caso del icono sólo se usaba para alinear o situar el icono en pantalla, en el caso de cursor su función es diferente, el punto activo indica el punto exacto de la acción del ratón, es el punto que se considera como la posición del cursor.

Los mensajes del ratón suelen referirse a un punto concreto, y ese punto es el punto activo del cursor.

## Crear cursores

Los cursores estándar no es necesario crearlos, ya que existen como parte del sistema. Podemos usar las funciones [LoadCursor](#) o [LoadImage](#) para obtener manipuladores de esos cursores. Las mismas funciones se usan para obtener manipuladores de cursores de recursos.

En el caso de cursores animados no es posible crearlos a partir de recursos, de modo que hay que cargarlos directamente desde un fichero ".ani" durante la ejecución, usando la función [LoadCursorFromFile](#), esta función también puede cargar ficheros de cursores no animados, con extensión ".cur".

```
HCURSOR caballo = LoadCursorFromFile("horse.ani");
```

También se pueden crear de forma dinámica mediante [CreateIconIndirect](#), y una estructura [ICONINFO](#), [GetIconInfo](#), pero es preferible usar cursores de recursos, ya que se evitan problemas derivados de la dependencia de dispositivo.

## Posición del cursor

La posición del cursor cambia como reflejo del movimiento del ratón, pero podemos cambiar su posición en cualquier momento usando la función [SetCursorPos](#). También, podemos recuperar la posición actual del cursor mediante [GetCursorPos](#).

Estas dos funciones trabajan con coordenadas de pantalla. Si queremos obtener o usar coordenadas de ventana para leer o modificar la posición del cursor podemos usar las funciones [ScreenToClient](#) o [ClientToScreen](#).

```
punto.x = 60;  
punto.y = 60;  
ClientToScreen(hwnd, &punto);  
SetCursorPos(punto.x, punto.y);
```

```
GetCursorPos(&punto);  
ScreenToClient(hwnd, &punto);
```

## Apariencia

Para obtener un manipulador del cursor actual se usa la función [GetCursor](#).

Para cambiar el cursor actual se usa la función [SetCursor](#), y sólo después de mover el cursor se mostrará la nueva apariencia. Recordemos que el sistema se encarga de mostrar siempre el cursor de acuerdo para la zona sobre la que esté, y cuando se sitúa en el área de cliente, se usa el cursor de la clase.

De modo que para que sea posible cambiar la apariencia del cursor, el cursor de la clase debe ser **NULL**. Pero esto significa que si movemos el cursor fuera del área de cliente, el cursor cambia automáticamente, y al regresar al área de cliente, mantiene el aspecto que tenía después de la última asignación de cursor. Por ejemplo, si situamos el cursor sobre el borde derecho se mostrará el cursor de doble flecha, este-oeste. Si volvemos al área de cliente, se mantiene ese cursor.

```
SetCursor(LoadCursor(hInstance, "flecha3D"));
```

Hay dos modos de evitar esto, uno es modificar el cursor de la clase, el otro, procesar el mensaje [WM\\_SETCURSOR](#).

## Modificar el cursor de clase

Ya vimos al principio del capítulo que es posible asignar un cursor para la clase de ventana, y que ese cursor se usará en el

área de cliente de todas las ventanas de esa clase. Ahora bien, es posible cambiar el cursor asociado a una clase, y en consecuencia, para todas las ventanas de dicha clase, esto se hace mediante la función [SetClassLong](#).

Esta función puede, en principio, cambiar cualquier valor de la estructura [WNDCLASS](#) o [WNDCLASSEX](#), pero en este caso nos limitaremos al cursor. Es tan sencillo como llamarla, usando como parámetros el manipulador de ventana, el índice correspondiente al cursor, que es [GCL\\_HCURSOR](#), y el manipulador del nuevo cursor de clase, convertido a un valor [LONG](#):

```
SetClassLong(hwnd, GCL_HCURSOR,  
              (LONG)LoadCursorFromFile("horse.ani"));
```

## El mensaje WM\_SETCURSOR

El mensaje [WM\\_SETCURSOR](#) nos permite un control mucho mayor sobre el aspecto del cursor, incluso aunque éste abandone el área de cliente. Se recibe cada vez que el cursor se mueve dentro de la ventana, y de ese modo podemos cambiar la apariencia del cursor a capricho.

Podemos, por ejemplo, cambiar el cursor dependiendo de la zona de la ventana, verificando si se encuentra dentro de un rectángulo o de una región determinada.

Si el cursor está fuera de cualquiera de las zonas de nuestro interés, probablemente queramos que su aspecto sea el esperado, por ejemplo, cuando esté sobre un borde o sobre la barra de menús. En este caso, debemos dejar que el mensaje se procese por el procedimiento por defecto:

```
case WM_SETCURSOR:  
    SetRect(&re, 20, 20, 80,80);  
    GetCursorPos(&punto);
```

```
ScreenToClient(hwnd, &punto);  
if(PtInRect(&re, punto))  
    SetCursor(LoadCursorFromFile("horse.ani"));  
else  
    return DefWindowProc(hwnd, msg, wParam,  
lParam);  
break;
```

En este ejemplo, si el cursor está sobre el rectángulo de coordenadas (20,20)-(80,80) se mostrará el cursor del caballo, en caso contrario, se ejecuta el proceso del mensaje por defecto, es decir: si el cursor está sobre el área de cliente, se mostrará el cursor de la clase, y en el área de no cliente, se mostrará el cursor que corresponda.

El problema con los cursores animados es que cada vez que se recibe el mensaje `WM_SETCURSOR` se vuelve a asignar el cursor, y éste vuelve a la primera imagen de la animación. Deberíamos crear un procedimiento que sólo lo cambie la primera vez que entramos en el área especificada, y no cada vez que se procese el mensaje, esto no sucede si cambiamos el cursor de clase.

## Ocultar y mostrar

También podemos ocultar o mostrar el cursor en cualquier momento, para ello usaremos la función `ShowCursor`. Esta función admite un parámetro de tipo `BOOL`, si es `TRUE`, el valor del contador se incrementa, si es `FALSE`, se decrementa. Si el valor del contador es mayor o igual que cero, el cursor se muestra, en caso contrario, se oculta.

La utilidad de ocultar el cursor es limitada, tal vez, evitar que el usuario lleve a cabo ciertas tareas que impliquen el uso de ratón en determinados momentos.

## Confinar el cursor

En ocasiones nos puede interesar que el cursor no salga de cierta zona de la pantalla hasta que se cumplan ciertas condiciones especiales, por ejemplo, podemos confinar el cursor a una zona concreta de una ventana, y limitar de este modo las posibilidades de mover el cursor, o de hacer clic en ciertas partes de la pantalla.

Para esto disponemos de la función [ClipCursor](#), que admite un parámetro de tipo [RECT](#) que define el rectángulo del que el cursor no puede salir. La función [GetClipCursor](#) permite recuperar ese rectángulo.

El cursor está asociado al ratón, y ambos son recursos únicos en el sistema, es decir, una aplicación no debería adueñarse de ellos de forma exclusiva. Cuando una aplicación que ha confinado el cursor pierde el foco, debe liberarlo para que pueda acceder a cualquier punto de la pantalla.

Esto se puede hacer procesando los mensajes [WM\\_SETFOCUS](#) y [WM\\_KILLFOCUS](#):

```
case WM_SETFOCUS:
    ClipCursor(NULL);
    break;
case WM_KILLFOCUS:
    GetWindowRect(hwnd, &re);
    ClipCursor(&re);
    break;
```

Este ejemplo confina el cursor a la ventana de la aplicación, e impide que el cursor se use fuera de ella. Si la aplicación pierde el foco, el cursor se libera, y cuando lo recupera, se vuelve a confinar.

Pero no bastará con esto, ya que cada vez que la ventana se mueva o cambie de tamaño, el cursor vuelve a quedar libre. Para evitar que esto suceda podemos recurrir a dos nuevos mensajes: [WM\\_SIZE](#) y [WM\\_MOVE](#), que se reciben cuando la ventana cambia de tamaño o de posición, respectivamente.



Sencillamente, procesaremos los mensajes [WM\\_SETFOCUS](#), [WM\\_SIZE](#) y [WM\\_MOVE](#) del mismo modo:

```
case WM_SETFOCUS:
    ClipCursor(NULL);
    break;
case WM_MOVE:
case WM_SIZE:
case WM_SETFOCUS:
    GetWindowRect(hwnd, &re);
    ClipCursor(&re);
    break;
```

## Destrucción de cursores

Cursores creados con [CreateIconIndirect](#) se destruyen con [DestroyCursor](#), no es necesario destruir el resto de los cursores.

## Ejemplo 33

# Capítulo 33 El ratón

Aunque importante, se considera que el ratón no es imprescindible en Windows, por lo tanto, debemos incluir todo lo necesario para que nuestras aplicaciones se puedan manejar exclusivamente con el teclado. Esta es la recomendación de Windows, sin embargo, no todo el mundo la sigue, y a menudo (cada vez más) encontramos aplicaciones que no pueden manejarse sin ratón.

Todas las entradas procedentes del ratón se reciben mediante mensajes. Así que si nuestra aplicación quiere procesar el ratón como una entrada, debe procesar esos mensajes.

Como vimos en el capítulo anterior, el ratón está asociado al cursor, cuando el primero se mueve, el segundo se desplaza en pantalla para indicar dicho movimiento. Tenemos esto tan asumido que frecuentemente decimos que movemos tanto el cursor como el ratón indistintamente. La ventana que recibe los mensajes del ratón es sobre la que se sitúa el punto activo del cursor (hotspot).

## Capturar el ratón

Como si fuésemos un gato, podemos capturar el ratón y mantenerlo cautivo para nuestra aplicación usando la función [SetCapture](#) e indicando qué ventana es la que captura el ratón. Para liberarlo se puede usar la función [ReleaseCapture](#), pero también se liberará si otra ventana captura el ratón o si el usuario hace clic en otra ventana distinta de la que lo ha capturado.

Cada vez que el ratón es capturado, se envía el mensaje [WM\\_CAPTURECHANGED](#) a la ventana que pierde la captura.

Un caso típico de captura de ratón es el del arrastre de objetos de una ventana a otra, por ejemplo, pulsamos el botón izquierdo del

ratón sobre el icono correspondiente a un fichero, y manteniéndolo pulsado movemos el cursor a otra ventana, sólo entonces soltamos el botón. Si queremos que la primera ventana siga recibiendo los mensajes del ratón aunque el cursor salga de sus límites, incluido el mensaje de soltar el botón, deberemos capturar el ratón.

No todos los mensajes sobre eventos del ratón son enviados a la ventana que lo ha capturado. Por ejemplo si el cursor se desplaza sobre ventanas diferentes a la que ha capturado el ratón, los mensajes sobre el movimiento del cursor se envían a esas ventanas, salvo que uno de los botones del ratón permanezca pulsado.

Otro efecto secundario destacable es que también perderemos las funciones normales del ratón sobre las ventanas hijas de la que ha capturado el ratón. Es decir, si capturamos el ratón, los clics sobre controles o menús de la ventana no realizan sus acciones habituales. De modo que no podremos acceder al menú, ni activar controles mediante el ratón.

## Configuración

Para saber si el ratón está presente se puede usar la función [GetSystemMetrics](#), con el parámetro [SM\\_MOUSEPRESENT](#). Además, podemos averiguar el número de botones del ratón, usando la misma función, con el parámetro [SM\\_CMOUSEBUTTONS](#). Se puede trabajar con ratones de uno, dos o tres botones. Los llamaremos izquierdo, derecho y central, y frecuentemente aparecerán con sus iniciales en inglés: L, R y M, respectivamente.

Las funciones de los botones izquierdo y derecho se pueden intercambiar cuando el usuario lo maneja con la mano izquierda (o si quiere hacerlo), mediante la función [SwapMouseButton](#). Esto significa que el botón izquierdo genera los mensajes del botón derecho, y viceversa.

Hay que tener en cuenta que el ratón es un recurso compartido, por lo tanto, esta modificación afectará a todas las ventanas.

## Mensajes

Cuando ocurre un evento relacionado con el ratón: pulsaciones de botones o movimientos, se envía un mensaje, y junto con él, las coordenadas del punto activo del cursor. Además, las ventanas siguen recibiendo estos mensajes aunque no tengan el foco del teclado. También los recibirán si la ventana ha capturado el ratón, aunque el cursor no esté sobre la ventana.

Los mensajes del ratón se envían del modo "post", es decir, son mensajes "lentos". En realidad se colocan en una cola que se procesa cuando el sistema tiene tiempo libre. Si se generan muchos mensajes en poco tiempo, el sistema elimina automáticamente los más antiguos, de modo que la aplicación sólo recibe los últimos.

Los mensajes "rápidos" se envían en modo "send", y el control pasa directamente al procedimiento de ventana, es decir, todos los mensajes de este tipo se procesan.

### Nota:

Lo siento, pero no he encontrado traducción para los términos "send" y "post" que indiquen todos los matices implícitos en inglés. Podríamos decir que los mensajes enviados "send" viajan por teléfono, el receptor los recibe tan pronto se generan, los del tipo "post" viajan por correo, y es posible que los últimos mensajes invaliden los anteriores o sencillamente, los hagan inútiles.

## Mensajes del área de cliente

Normalmente sólo procesaremos estos mensajes, e ignoraremos el resto.

El mensaje `WM_MOUSEMOVE` se recibe cada vez que el usuario mueve el cursor sobre la ventana. Este mensaje es el que más frecuentemente se elimina, ya que puede ser generado muchas veces por segundo.

Existen otros mensajes, uno por cada evento de pulsar o soltar un botón, o por un doble clic, y para cada uno de estos tres eventos, variantes para cada uno de los tres botones del ratón. En total nueve mensajes.

`WM_LBUTTONDOWN`, `WM_MBUTTONDOWN` y `WM_RBUTTONDOWN` se envían cada vez que el usuario pulsa el botón izquierdo, central o derecho, respectivamente.

`WM_LBUTTONUP`, `WM_MBUTTONUP` y `WM_RBUTTONUP` se envían cuando el usuario suelta cada uno de los botones izquierdo, central o derecho, respectivamente.

`WM_LBUTTONDBLCLK`, `WM_MBUTTONDBLCLK` y `WM_RBUTTONDBLCLK` se envían cuando el usuario hace doble clic sobre el botón izquierdo, central o derecho respectivamente. En estos casos también se envían los mensajes correspondientes a las pulsaciones y sueltas individuales que completan el doble clic. Por ejemplo, un mensaje `WM_LBUTTONDBLCLK`, se recibe dentro de una secuencia de mensajes `WM_LBUTTONDOWN`, `WM_LBUTTONUP`, `WM_LBUTTONDBLCLK` y `WM_LBUTTONUP`.

Para que se genere un mensaje de doble clic se deben cumplir ciertos requisitos. El segundo clic debe producirse en un área determinada alrededor del primero, y dentro de un intervalo de tiempo determinado. Tanto las dimensiones del área (en realidad un rectángulo), como el tiempo de doble clic se pueden modificar mediante funciones del API, pero es mejor dejar este trabajo al usuario mediante Panel de Control, ya que estos cambios afectan a todas las ventanas.

Las ventanas no reciben los mensajes de doble clic por defecto, hay que activar un estilo determinado para que esto sea así. En

concreto, al ventana debe crearse a partir de una clase que tenga el estilo [CS\\_DBLCLKS](#). Hasta ahora siempre hemos creado ventanas de este tipo en nuestros ejemplos:

```
WNDCLASSEX wincl;          /* Estructura de datos para la
clase de ventana */

/* Estructura de la ventana */
wincl.hInstance = hThisInstance;
wincl.lpszClassName = "NUESTRA_CLASE";
wincl.lpfnWndProc = WindowProc;      /* Esta
función es invocada por Windows */
wincl.style = CS_DBLCLKS;            /* Captura los
doble-clicks */
wincl.cbSize = sizeof (WNDCLASSEX);

/* Usar icono y puntero por defecto */
wincl.hIcon = LoadIcon (hThisInstance, "Icono");
wincl.hIconSm = LoadIcon (hThisInstance, "Icono");
wincl.hCursor = NULL;
wincl.lpszMenuName = "Menu";
wincl.cbClsExtra = 0;      /* Sin información adicional
para la */
wincl.cbWndExtra = 0;      /* clase o la ventana */
/* Usar el color de fondo por defecto para es escritorio
*/
wincl.hbrBackground =
GetSysColorBrush(COLOR_BACKGROUND);

/* Registrar la clase de ventana, si falla, salir del
programa */
if(!RegisterClassEx(&wincl)) return 0;
```

En todos los casos, en el parámetro `lParam` de cada mensaje se reciben las coordenadas del punto activo del cursor. En la palabra de menor peso la coordenada `x` y en la de mayor peso, la coordenada `y`. Además, las coordenadas son coordenadas de cliente, es decir, relativas a la esquina superior izquierda del área de cliente. Para separar estos valores se puede usar la macro [MAKEPOINTS](#), que convierte el valor en el parámetro `lParam` en una estructura [POINTS](#).

También en todos los casos, el parámetro wParam del mensaje contiene información sobre si ciertas teclas o botones del ratón están pulsados.

Valor	Descripción
<b>MK_CONTROL</b>	Activo si la tecla CTRL está pulsada.
<b>MK_LBUTTON</b>	Activo si el botón izquierdo del ratón está pulsado.
<b>MK_MBUTTON</b>	Activo si el botón central del ratón está pulsado.
<b>MK_RBUTTON</b>	Activo si el botón derecho del ratón está pulsado.
<b>MK_SHIFT</b>	Activo si la tecla MAYÚSCULAS está pulsada.

```
static int izq, cen, der;
static POINTS punto;
...
case WM_MOUSEMOVE:
    punto = MAKEPOINTS(lParam);
    izq = (wParam & MK_LBUTTON) ? 1 : 0;
    cen = (wParam & MK_MBUTTON) ? 1 : 0;
    der = (wParam & MK_RBUTTON) ? 1 : 0;
    hdc = GetDC(hwnd);
    Pintar(hdc, izq, cen, der, punto, TRUE);
    ReleaseDC(hwnd, hdc);
    break;
case WM_LBUTTONDOWN:
    izq = 2;
    punto = MAKEPOINTS(lParam);
    hdc = GetDC(hwnd);
    Pintar(hdc, izq, cen, der, punto, TRUE);
    ReleaseDC(hwnd, hdc);
    break;
case WM_LBUTTONUP:
    izq = 3;
    punto = MAKEPOINTS(lParam);
    hdc = GetDC(hwnd);
    Pintar(hdc, izq, cen, der, punto, TRUE);
    ReleaseDC(hwnd, hdc);
    break;
```

```
case WM_LBUTTONDOWNBLCLK:
    izq = 4;
    punto = MAKEPOINTS(lParam);
    hdc = GetDC(hwnd);
    Pintar(hdc, izq, cen, der, punto, TRUE);
    ReleaseDC(hwnd, hdc);
    break;
```

## Mensajes del área de no cliente

Existe la misma gama de mensajes de ratón para el área de no cliente que para el área de cliente, el nombre de los mensajes es el mismo, pero con la letras NC después del '\_', por ejemplo, el mensaje que notifica movimientos del cursor dentro del área de no cliente es [WM\\_NCMOUSEMOVE](#).

Del mismo modo, los mensajes relacionados con clics de botones son:

[WM\\_NCLBUTTONDOWN](#), [WM\\_NCMBUTTONDOWN](#) y [WM\\_NCRBUTTONDOWN](#) se envían cada vez que el usuario pulsa el botón izquierdo, central o derecho, respectivamente.

[WM\\_NCLBUTTONUP](#), [WM\\_NCMBUTTONUP](#) y [WM\\_NCRBUTTONUP](#) se envían cuando el usuario suelta cada uno de los botones izquierdo, central o derecho, respectivamente.

[WM\\_NCLBUTTONDOWNBLCLK](#), [WM\\_NCMBUTTONDOWNBLCLK](#) y [WM\\_NCRBUTTONDOWNBLCLK](#) se envían cuando el usuario hace doble clic sobre el botón izquierdo, central o derecho respectivamente. En estos casos también se envían los mensajes correspondientes a las pulsaciones y sueltas individuales que completan el doble clic. Por ejemplo, un mensaje [WM\\_NCLBUTTONDOWNBLCLK](#), se recibe dentro de una secuencia de mensajes [WM\\_NCLBUTTONDOWN](#), [WM\\_NCLBUTTONUP](#), [WM\\_NCLBUTTONDOWNBLCLK](#) y [WM\\_NCLBUTTONUP](#).

Estos mensajes se deben procesar con cuidado, ya que al hacer que la aplicación responda a ellos podremos perder muchas de las funciones propias del área de no cliente, como acceso a menús, movimiento de la ventana, cambio de tamaño, etc, salvo que



llamemos al procedimiento de ventana por defecto después de procesar el mensaje. Pero generalmente no tendremos necesidad de procesar estos mensajes.

En el caso de estos mensajes, el parámetro *wParam* no contiene información sobre el estado de teclas y botones, sino sobre el *hit test*, la zona en la que se encuentra el punto activo del cursor. De este modo podemos saber si está sobre un borde, un menú, la barra de título, etc. Veremos esto en más detalle más abajo, al ver el mensaje [WM\\_NCHITTEST](#).

El parámetros *lParam* sigue conteniendo las coordenadas del cursor, pero en coordenadas de pantalla.

```
case WM_NCMOUSEMOVE:
    punto = MAKEPOINTS(lParam);
    izq = cen = der = 0;
    hdc = GetDC(hwnd);
    Pintar(hdc, izq, cen, der, punto, FALSE);
    ReleaseDC(hwnd, hdc);
    break;
case WM_NCLBUTTONDOWN:
    izq = 2;
    punto = MAKEPOINTS(lParam);
    hdc = GetDC(hwnd);
    Pintar(hdc, izq, cen, der, punto, FALSE);
    ReleaseDC(hwnd, hdc);
    return DefWindowProc(hwnd, msg, wParam, lParam);
    break;
case WM_NCLBUTTONUP:
    izq = 3;
    punto = MAKEPOINTS(lParam);
    hdc = GetDC(hwnd);
    Pintar(hdc, izq, cen, der, punto, FALSE);
    ReleaseDC(hwnd, hdc);
    return DefWindowProc(hwnd, msg, wParam, lParam);
    break;
case WM_NCLBUTTONDBLCLK:
    izq = 4;
    punto = MAKEPOINTS(lParam);
    hdc = GetDC(hwnd);
    Pintar(hdc, izq, cen, der, punto, FALSE);
    ReleaseDC(hwnd, hdc);
```

```
return DefWindowProc(hwnd, msg, wParam, lParam);  
break;
```

## Mensaje WM\_NCHITTEST

El mensaje [WM\\_NCHITTEST](#) se envía a la ventana que contiene el cursor, o a la que ha capturado el ratón, cada vez que ocurre un evento del ratón. Este mensaje se usa por Windows para determinar si se debe enviar un nuevo mensaje de área de cliente o de área de no cliente. Si queremos que nuestra aplicación reciba mensajes del ratón debemos dejar que la función [DefWindowProc](#) procese este mensaje.

```
static POINTS puntoHT;  
static LRESULT hittest;  
...  
case WM_NCHITTEST:  
    puntoHT = MAKEPOINTS(lParam);  
    hdc = GetDC(hwnd);  
    sprintf(cad, "Punto HIT-TEST: (%4d, %4d)",  
        puntoHT.x, puntoHT.y);  
    TextOut(hdc, 10, 110, cad, strlen(cad));  
    MostrarHitTest(hdc, hittest);  
    ReleaseDC(hwnd, hdc);  
    hittest = DefWindowProc(hwnd, msg, wParam,  
lParam);  
    return hittest;
```

En el parámetro *lParam* se reciben las coordenadas del punto activo del cursor en coordenadas de pantalla:

Cuando la función [DefWindowProc](#) procesa este mensaje devuelve un código de hit-test, que depende de la zona en que se encuentre el cursor.

Valor	Posición del punto activo
HTBORDER	En el borde de la ventana que no tiene borde de cambio de tamaño.

HTBOTTOM	En borde horizontal inferior de una ventana.
HTBOTTOMLEFT	En la esquina inferior izquierda del borde de una ventana.
HTBOTTOMRIGHT	En la esquina inferior derecha del borde de una ventana.
HTCAPTION	En una barra de título.
HTCLIENT	En un área de cliente.
HTERROR	En el fondo de la pantalla o en una línea de división entre ventanas (lo mismo que HTNOWHERE, excepto que DefWindowProc produce un pitido de sistema para indicar un error).
HTGROWBOX	En una caja de cambio de tamaño (lo mismo que HTSIZE).
HTHSCROLL	En la barra de desplazamiento horizontal.
HTLEFT	En el borde izquierdo de una ventana.
HTMENU	En un menú.
HTNOWHERE	En el fondo de la pantalla o en una línea de división entre ventanas.
HTREDUCE	En un botón de minimizar.
HTRIGHT	En el borde derecho de una ventana.
HTSIZE	En una caja de cambio de tamaño (lo mismo que HTGROWBOX).
HTSYSMENU	En un menú de sistema o en un botón de cierre en una ventana hija.

HTTOP	En el borde horizontal superior de una ventana.
HTTOPLEFT	En la esquina superior izquierda del borde de una ventana.
HTTOPRIGHT	En la esquina superior derecha de un borde de ventana.
HTTRANSPARENT	En una ventana actualmente tapada por otra ventana.
HTVSCROLL	En la barra de desplazamiento vertical.
HTZOOM	En un botón de maximizar.

Si el valor devuelto por el procedimiento de ventana es **HTCLIENT** es porque está en el área de cliente, en ese caso las coordenadas se trasladan a coordenadas de cliente y se envía un mensaje "lento" de área de cliente, y en el parámetro *wParam* se envía el estado de los botones del ratón. Si se encuentra en otra zona, se envía un mensaje "lento" de área de no cliente, se mantienen las coordenadas en coordenadas de pantalla y en el parámetro *wParam* se envía el código hit-test.

## Mensaje WM\_MOUSEACTIVATE

El mensaje **WM\_MOUSEACTIVATE** se envía a una ventana cuando se hace clic con uno de los botones del ratón cuando el cursor está sobre ella o sobre una de sus ventanas hijas, y si la ventana está inactiva. Este mensaje se envía después del mensaje **WM\_NCHITTEST** y antes de cualquier mensaje de área de cliente o de área de no cliente.

Si se deja que **DefWindowProc** procese este mensaje, se activará la ventana y se enviará el mensaje de pulsación de botón a la ventana.

Si procesamos el mensaje en nuestro procedimiento de ventana tenemos más opciones, y podemos controlar si se activa o no la ventana y si se descarta el mensaje de pulsación del botón o no.

Podemos devolver los siguientes valores para conseguir estos resultados al procesar este mensaje:

Valor	Significado
MA_ACTIVATE	Activa la ventana, y no descarta el mensaje de ratón.
MA_ACTIVATEANDEAT	Activa la ventana, y descarta el mensaje de ratón.
MA_NOACTIVATE	No activa la ventana, y no descarta el mensaje de ratón.
MA_NOACTIVATEANDEAT	No activa la ventana, pero descarta el mensaje de ratón.

## Otros mensajes de ratón

Algunas de las características que ahora son corrientes en el manejo del ratón, no lo eran o ni siquiera existían hace unos años. Por ejemplo, la rueda del ratón es un invento relativamente reciente. Además, debido a los navegadores de Internet, se han popularizado algunos eventos relacionados con el ratón que antes no existían, como el paso sobre zonas determinadas, o la transición entre unas zonas y otras de la pantalla. Estos eventos se usan en los navegadores para cambiar el aspecto de textos o botones, y de ese modo llamar la atención del usuario sobre ellos para indicar que son zonas activas a clics.

Veamos ahora estos nuevos mensajes en el API.

### Mensaje WM\_MOUSEWHEEL (Windows NT)

El mensaje [WM\\_MOUSEWHEEL](#) se envía cada vez que el usuario activa la rueda de desplazamiento del ratón. Los parámetros

de este mensaje son los mismos que en el mensaje [WM\\_MOUSEMOVE](#), pero para incluir la información sobre el avance o retroceso de la rueda, el parámetro *wParam* se ha dividido en dos. En la parte baja se empaquetan las banderas sobre el estado de los botones, y en la parte alta el valor de avance o retroceso de la rueda. Usaremos las macros [LOWORD](#) y [HIWORD](#) para extraer esos valores.

Los valores de avance y retroceso de la rueda serán siempre múltiplos de 120. Esto se ha hecho así para permitir que en el futuro se puedan construir dispositivos compatibles con la rueda, pero que proporcionen mayor precisión. Debemos considerar siempre que una unidad de desplazamiento de la rueda es 120, o para evitar problemas de dependencias, de la constante **WHEEL\_DELTA**.

Los valores positivos indican movimientos de rueda hacia adelante, y los negativos, hacia atrás.

```
case WM_MOUSEWHEEL:
    punto = MAKEPOINTS(lParam);
    izq = (LOWORD(wParam) & MK_LBUTTON) ? 1 : 0;
    cen = (LOWORD(wParam) & MK_MBUTTON) ? 1 : 0;
    der = (LOWORD(wParam) & MK_RBUTTON) ? 1 : 0;
    rueda = HIWORD(wParam);
    rueda /= WHEEL_DELTA;
    hdc = GetDC(hwnd);
    Pinta(hdc, izq, cen, der, punto, TRUE);
    sprintf(cad, "rueda = %04d ", rueda);
    TextOut(hdc, 10, 130, cad, strlen(cad));
    ReleaseDC(hwnd, hdc);
    break;
```

## Trazar eventos del ratón (Windows NT)

Nuestra aplicación puede recibir dos mensajes sobre eventos del ratón: [WM\\_MOUSELEAVE](#) y [WM\\_MOUSEHOVER](#), cuando el ratón abandona una ventana o cuando permanece sobre un área determinada de la ventana durante un periodo de tiempo,

respectivamente. Pero para que esto suceda, previamente debemos activar el trazado de eventos, mediante una llamada a la función [TrackMouseEvent](#).

Esta función necesita como parámetro un puntero a una estructura [TRACKMOUSEEVENT](#). En esa estructura indicamos qué eventos queremos que se notifique, y el tiempo necesario para generar un mensaje [WM\\_MOUSEHOVER](#):

```
case WM_NCHITTEST:
    hittest = DefWindowProc(hwnd, msg, wParam,
lParam);
    if(HTCLIENT == hittest) {
        if(!dentro) {
            dentro = TRUE;
            tme.cbSize = sizeof(TRACKMOUSEEVENT);
            tme.dwFlags = TME_HOVER | TME_LEAVE;
            tme.hwndTrack = hwnd;
            tme.dwHoverTime = 1000;
            TrackMouseEvent(&tme);
        }
    }
    return hittest;
break;
```

En este ejemplo llamamos a [TrackMouseEvent](#) para que nos notifique ambos eventos para la ventana *hwnd*, y ajustamos el tiempo *Hover* en un segundo.

## Mensaje WM\_MOUSELEAVE (Windows NT)

Si hemos activado el evento *Leave* recibiremos un mensaje [WM\\_MOUSELEAVE](#) cuando el cursor abandone el área de cliente de la ventana. Una vez que se recibe el mensaje no se vuelve a generar, salvo que volvamos a activar el evento, usando de nuevo la función [TrackMouseEvent](#).

```
case WM_MOUSELEAVE:
    dentro = FALSE;
    hdc = GetDC(hwnd);
    TextOut(hdc, 10, 170, "Leave", 5);
    ReleaseDC(hwnd, hdc);
    break;
```

## Mensaje WM\_MOUSEHOVER (Windows NT)

Del mismo modo, si hemos activado el evento *Hover* recibiremos un mensaje [WM\\_MOUSEHOVER](#) cuando haya transcurrido el tiempo indicado y el cursor no se haya movido de una zona determinada del área de cliente de la ventana. Esta zona está predeterminada por Windows, aunque se puede modificar su tamaño (ver [TRACKMOUSEEVENT](#)). Una vez que se recibe el mensaje no se vuelve a generar, salvo que volvamos a activar el evento, usando de nuevo la función [TrackMouseEvent](#).

```
case WM_MOUSEHOVER:
    hdc = GetDC(hwnd);
    TextOut(hdc, 10, 170, "Hover", 5);
    ReleaseDC(hwnd, hdc);
    break;
```

## Ejemplo 34

### Arrastrar objetos

Una de las operaciones más frecuentes que se realizan mediante el ratón es la de arrastrar objetos. No entraremos en muchos detalles por ahora, Windows dispone de formas especiales de realizar arrastre de objetos entre distintas ventanas y



aplicaciones, pero de momento veremos un ejemplo sencillo sobre cómo arrastrar objetos dentro de una misma ventana.

En este ejemplo usaremos iconos como objetos. A cada icono le corresponde una imagen y una posición en pantalla:

```
typedef struct {
    HICON icono;
    POINT coordenada;
} Objeto;
```

Al procesar el mensaje `WM_CREATE` inicializamos el array de objetos:

```
case WM_CREATE:
    hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
    objeto[0].icono = LoadIcon(hInstance, "ufo");
    objeto[0].coordenada.x = 10;
    objeto[0].coordenada.y = 10;
    objeto[1].icono = LoadIcon(hInstance, "libro");
    objeto[1].coordenada.x = 10;
    objeto[1].coordenada.y = 50;
    objeto[2].icono = LoadIcon(hInstance, "mundo");
    objeto[2].coordenada.x = 10;
    objeto[2].coordenada.y = 90;
    objeto[3].icono = LoadIcon(hInstance,
    "hamburguesa");
    objeto[3].coordenada.x = 50;
    objeto[3].coordenada.y = 10;
    objeto[4].icono = LoadIcon(hInstance, "smile");
    objeto[4].coordenada.x = 50;
    objeto[4].coordenada.y = 50;
    capturado = -1;
    break;
```

Una operación de arrastre comienza con un clic sobre un objeto. Después, sin soltar el botón del ratón, se desplaza el ratón, y con él el objeto, a la posición deseada, y finalmente, se suelta el botón del ratón en esa posición.

Lo primero es seleccionar el objeto a arrastrar. Para ello procesaremos el mensaje `WM_LBUTTONDOWN`, y comprobaremos si las coordenadas del cursor corresponden con alguno de los objetos que es posible arrastrar. Si es así, guardamos en una variable el identificador del objeto, y al mismo tiempo, activamos el modo de arrastre:

```
case WM_LBUTTONDOWN:
    punto = MAKEPOINTS(lParam);
    for(i = 0; capturado == -1 && i < 5; i++) {
        SetRect(&re,
            objeto[i].coordenada.x,
            objeto[i].coordenada.y,
            objeto[i].coordenada.x+32,
            objeto[i].coordenada.y+32);
        POINTSTOPOINT(lpunto, punto);
        if(PtInRect(&re, lpunto)) {
            capturado = i;
            ClientToScreen(hwnd,
                &objeto[i].coordenada);
            SetCursorPos(objeto[i].coordenada.x,
                objeto[i].coordenada.y);
            SetCapture(hwnd);
            ShowCursor(FALSE);
        }
    }
    break;
```

Si la variable *capturado* vale -1, indica que no se está realizando un arrastre, si tiene otro valor, indica el objeto arrestrado.

Para cada objeto calculamos las coordenadas de un rectángulo que lo contiene, y comprobamos si la posición actual del cursor está dentro del rectángulo. Si es así, actualizamos el valor de *capturado*, cambiamos la posición del cursor a la esquina superior izquierda del objeto, y ocultamos el cursor.

El cambio de coordenadas sirve para eliminar el salto que se produciría cuando pinchemos sobre un punto distinto de la esquina

superior izquierda. Ocultar el cursor hace que el objeto arrastrado parezca sustituir al cursor, y hace más fácil arrastrarlo con precisión.

Además, capturamos el ratón para que la operación de arrastre no se interrumpa si el cursor sale del área de cliente de la ventana.

Durante el arrastre debemos procesar el mensaje `WM_MOUSEMOVE`. Cada vez que recibamos el mensaje, borraremos el objeto en su posición actual, actualizamos las coordenadas según la posición del cursor, y volvemos a dibujarlo en la nueva posición:

```
case WM_MOUSEMOVE:
    punto = MAKEPOINTS(lParam);
    if(capturado != -1) {
        hdc = GetDC(hwnd);
        //drag
        // Borrar en posición actual:
        SetRect(&re,
            objeto[capturado].coordenada.x,
            objeto[capturado].coordenada.y,
            objeto[capturado].coordenada.x+32,
            objeto[capturado].coordenada.y+32);
        FillRect(hdc, &re,
            GetSysColorBrush(COLOR_BACKGROUND));
        // Actualizar coordenadas:
        POINTSTOPOINT(objeto[capturado].coordenada,
punto);

        // Pintar en nueva posición:
        DrawIcon(hdc,
            objeto[capturado].coordenada.x,
            objeto[capturado].coordenada.y,
            objeto[capturado].icono);
        ReleaseDC(hwnd, hdc);
        InvalidateRect(hwnd, &re, FALSE);
    }
    break;
```

Para borrar pintamos usando el color de fondo, la zona ocupada por el objeto. Para actualizar la posición del objeto usamos la macro `POINTSTOPOINT`, esto es porque la posición del cursor se almacena en una estructura `POINTS`, y la del objeto en una

estructura **POINT**. A continuación mostramos el objeto, e invalidamos la zona que ocupaba originalmente. Esto último es necesario, ya que el objeto arrastrado puede pasar sobre otros objetos borrándolos.

Finalmente, cuando soltemos el botón se recibirá un mensaje **WM\_LBUTTONDOWN**. En ese momento debemos dar por terminada la operación de arrastre, y regresaremos al estado inicial:

```
case WM_LBUTTONDOWN:
    if(capturado != -1) {
        capturado = -1;
        InvalidateRect(hwnd, NULL, FALSE);
        ReleaseCapture();
        ShowCursor(TRUE);
    }
    break;
```

Asignamos -1 a *capturado*, redibujamos toda la ventana, liberamos el ratón y mostramos el cursor.

## Ejemplo 35

# Capítulo 34 El Teclado

Al igual que el ratón, las entradas del teclado se reciben en forma de mensajes. En este capítulo veremos el manejo básico del teclado, y algunas características relacionadas con este dispositivo.

Como pasa con otros dispositivos del ordenador, en el teclado distinguimos al menos dos capas: la del dispositivo físico y la del dispositivo lógico.

En cuanto al dispositivo físico, el teclado no es más que un conjunto de teclas. Cada una de ellas genera un código diferente, cada vez que es pulsada o liberada, a esos códigos los llamaremos códigos de escaneo (*scan codes*). Por supuesto, dado que estamos en la capa física, estos códigos son dependientes del dispositivo, y en principio, cambiarán dependiendo del fabricante del teclado.

Pero Windows nos permite hacer nuestros programas independientes del dispositivo, de modo que no será frecuente que tengamos que trabajar con códigos de escaneo, y aunque el API nos informe de esos códigos, generalmente los ignoraremos.

En la capa lógica, el driver del teclado traduce los códigos de escaneo a códigos de tecla virtual (*virtual-key codes*). Estos códigos son independientes del dispositivo, e identifican el propósito de cada tecla. Generalmente, serán esos los códigos que usemos en nuestros programas. (Tabla al final).

## El Foco del teclado

Los mensajes del teclado se envían al proceso de primer plano que haya creado la ventana que actualmente tiene el foco del teclado. El teclado se comparte entre todas las ventanas abiertas, pero sólo una de ellas puede poseer el foco del teclado, los

mensajes del teclado llegan a esa ventana a través del bucle de mensajes del proceso que las creó.

Sólo una ventana, o ninguna, puede poseer el foco del teclado, para averiguar cual es la que lo posee podemos usar la función [GetFocus](#), siempre que esa ventana pertenezca al proceso actual. Para asignar el foco a la ventana que queramos se usa la función [SetFocus](#). Ya hemos usado esta función anteriormente, para cambiar el control que recibe la entrada del usuario al iniciar un cuadro de diálogo.

Cuando una ventana pierde el foco, recibe un mensaje [WM\\_KILLFOCUS](#) y a la ventana que lo recibe, se le envía un mensaje [WM\\_SETFOCUS](#). En ocasiones se puede usar el mensaje [WM\\_KILLFOCUS](#) para realizar la validación de los datos de un control, o cualquier tarea, antes de perder definitivamente la atención del usuario. Del mismo modo, el mensaje [WM\\_SETFOCUS](#) se puede usar para preparar una ventana o control antes de que el usuario pueda modificar los datos que contiene. Normalmente, si una ventana acepta entradas desde el teclado, sabremos si tiene el foco porque se muestra un caret en su interior.

Una propiedad relacionada con el foco del teclado es el de ventana activa. La ventana activa es con la que el usuario está trabajando. La ventana con el foco del teclado es, o bien la ventana activa, o bien una de sus ventanas hija. La ventana activa se distingue porque su barra de título cambia de color, y porque suele tener el foco del teclado y del ratón (aunque esto no siempre es cierto).

El usuario puede cambiar de ventana activa, usando el teclado, haciendo clic sobre ella, etc. También es posible hacerlo usando la función [SetActiveWindow](#) y con [GetActiveWindow](#), un proceso puede obtener el manipulador de la ventana activa asociada, si es que existe. Cada vez que una ventana deja de ser la activa, recibe un mensaje [WM\\_ACTIVATE](#), y después se envía el mismo mensaje a la que pasa a ser activa.

```

        case WM_ACTIVATE:
            if((HWND)lParam == NULL) strcpy(nombre, "NULL");
            else GetWindowText((HWND)lParam, nombre, 128);
            if(LOWORD(wParam) == WA_INACTIVE)
                sprintf(cad, "Ventana desactivada hacia %s",
nombre);
            else
                sprintf(cad, "Ventana activada desde %s",
nombre);
            break;
        case WM_SETFOCUS:
            if((HWND)wParam == NULL) strcpy(nombre, "NULL");
            else GetWindowText((HWND)wParam, nombre, 128);
            sprintf(cad, "Pérdida de foco en favor de %s",
nombre);
            break;
        case WM_SETFOCUS:
            if((HWND)wParam == NULL) strcpy(nombre, "NULL");
            else GetWindowText((HWND)wParam, nombre, 128);
            sprintf(cad, "Foco recuperado desde %s", nombre);
            break;

```

## Ventanas inhibidas

A veces es útil hacer que una ventana no pueda recibir el foco del teclado, ya sea porque los datos que contiene no deban ser modificados, o porque no tengan sentido en un contexto determinado. En ese caso, podemos inhibir tal ventana usando la función [EnableWindow](#). La misma función se usa para desinhibirla. Una ventana inhibida no puede recibir mensajes del teclado ni del ratón.

```

    static BOOL cambio;
    ...
    cambio = FALSE;
    EnableWindow(GetDlgItem(hDlg, ID_CONTROL1),
!cambio);
    EnableWindow(GetDlgItem(hDlg, ID_CONTROL2),

```

```
cambio);  
SetFocus(GetDlgItem(hDlg, ID_CONTROL1));
```

## Ejemplo 36

### Mensajes de pulsación de teclas

La acción de pulsar una tecla implica dos eventos, uno cuando se pulsa y otro cuando se libera. Cuando se pulsa una tecla se envía un mensaje `WM_KEYDOWN` o `WM_SYSKEYDOWN` a la ventana que tiene el foco del teclado, y cuando se libera, un mensaje `WM_KEYUP` o `WM_SYSKEYUP`.

Los mensajes `WM_SYSKEYDOWN` y `WM_SYSKEYUP` se refieren a teclas de sistema. Las teclas de sistema son las que se pulsan manteniendo pulsada la tecla [Alt]. Los otros dos mensajes se refieren a teclas que no sean de sistema.

En todos los casos, el parámetro *wParam* contiene el código de tecla virtual, y el parámetro *lParam* varios datos asociados a la tecla, como repeticiones, código de escaneo, si se trata de una tecla extendida, el código de contexto, el estado previo de la tecla y el estado de transición.

Podemos crear un campo de bits para tratar estos datos más fácilmente:

```
typedef union {  
    struct {  
        unsigned int repeticion:16;  
        unsigned int scan:8;  
        unsigned int extendida:1;  
        unsigned int reservado:4;  
        unsigned int contexto:1;  
        unsigned int previo:1;  
        unsigned int transicion:1;  
    };  
    unsigned int lParam;  
} keyData;
```



---

Cuando el usuario deja pulsada una tecla generalmente tiene la intención de repetir varias veces esa pulsación. El sistema está preparado para ello, y a partir de cierto momento, se generará una repetición cada cierto intervalo de tiempo. Los dos tiempos se pueden ajustar en el Panel de control.

Pero lo que nos interesa en este caso es que el sistema genera nuevos mensajes `WM_KEYDOWN` o `WM_SYSKEYDOWN`, sin los correspondientes mensajes de tecla liberada. Es más, cada uno de los mensajes puede corresponder a una pulsación, si el sistema es lo bastante rápido para procesar cada pulsación individual; o a varias, si se acumulan repeticiones entre dos mensajes consecutivos.

Para saber cuantas repeticiones de tecla están asociadas a un mensaje de tecla pulsada hay que examinar el campo de repetición del parámetro *lParam*.

El código de escaneo, como comentamos antes, es dependiente del dispositivo, y por lo tanto, generalmente no tiene utilidad para nosotros.

El bit de tecla extendida indica si se trata de una tecla específica de un teclado extendido. Generalmente, los ordenadores actuales siempre usan un teclado extendido.

El bit de contexto siempre es cero en los mensajes `WM_KEYDOWN` y `WM_KEYUP`, en el caso de los mensajes `WM_SYSKEYDOWN` y `WM_SYSKEYUP` será 1 si la tecla Alt está pulsada.

El bit de estado previo indica si la tecla estaba pulsada antes de enviar el mensaje, 1 si lo estaba, 0 si no lo estaba.

Y el bit de transición siempre es 0 en el caso de mensajes `WM_KEYDOWN` y `WM_SYSKEYDOWN`, y 1 en el caso de `WM_KEYUP` y `WM_SYSKEYUP`.

Cuando nuestra aplicación necesite procesar los mensajes de pulsación de tecla de sistema, debemos tener cuidado de pasarlos a la función `DefWindowProc` para que se procesen por el sistema. No

lo hacemos esto, nuestra aplicación no responderá al menú desde el teclado, mediante combinaciones Alt+tecla.

Los mensajes de pulsación de tecla se usarán cuando queramos tener un control bastante directo del teclado, generalmente no nos interesa tanto control, y los mensajes de carácter serán suficientes.

```
case WM_PAINT:
    hdc = BeginPaint(hwnd, &ps);
    SetBkColor(hdc, GetSysColor(COLOR_BACKGROUND));
    for(i = 0; i < nLineas; i++)
        TextOut(hdc, 10, i*20, lista[i],
strlen(lista[i]));
    EndPaint(hwnd, &ps);
    break;
case WM_KEYDOWN:
    for(i = nLineas; i > 0; i--)
        strcpy(lista[i], lista[i-1]);
    if(nLineas < 39) nLineas++;
    kd.lParam = lParam;
    sprintf(lista[0], "Tecla %d pulsada Rep=%d "
        "[Ext:%d Ctx:%d Prv:%d Trn:%d]",
        (int)wParam, kd.repeticion, kd.extendida,
        kd.contexto, kd.previo, kd.transicion);
    InvalidateRect(hwnd, NULL, TRUE);
    break;
case WM_KEYUP:
    for(i = nLineas; i > 0; i--)
        strcpy(lista[i], lista[i-1]);
    if(nLineas < 39) nLineas++;
    kd.lParam = lParam;
    sprintf(lista[0], "Tecla %d liberada "
        "[Ext:%d Ctx:%d Prv:%d Trn:%d]",
        (int)wParam, kd.extendida,
        kd.contexto, kd.previo, kd.transicion);
    InvalidateRect(hwnd, NULL, TRUE);
    break;
```

## Nombres de teclas

Una función que puede resultar útil en algunas circunstancias es [GetKeyNameText](#), que nos devuelve el nombre de una tecla. Como

parámetros sólo necesita el parámetro *lParam* entregado por un mensaje de pulsación de tecla, un búffer para almacenar el nombre y el tamaño del búffer:

```
char texto[128], cad[64];
...
case WM_KEYDOWN:
    GetKeyNameText(lParam, cad, 64);
    sprintf(texto, "Tecla %s (%d) pulsada",
        cad, (int)wParam);
    break;
```

## El bucle de mensajes

Es el momento de comentar algo sobre el bucle de mensajes que estamos usando desde el principio de este texto:

```
while(TRUE == GetMessage(&mensaje, NULL, 0, 0))
{
    /* Traducir mensajes de teclas virtuales a mensajes
de caracteres */
    TranslateMessage(&mensaje);
    /* Enviar mensaje al procedimiento de ventana */
    DispatchMessage(&mensaje);
}
```

Me refiero a la función [TranslateMessage](#), que como dice el comentario, traduce los mensajes de pulsaciones de teclas a mensajes de carácter. Si nuestra aplicación debe procesar los mensajes de pulsaciones de teclas no debería llamar a esta función en el bucle de mensajes. De todos modos, los mensajes de pulsación de teclas parecen llegar en los dos casos, pero no es mala idea seguir la recomendación del API en este caso.

## Ejemplo 37

# Mensajes de carácter

Si usamos la función `TranslateMessage`, cada mensaje `WM_KEYDOWN` se traduce en un mensaje `WM_CHAR` o `WM_DEADCHAR`; y cada mensaje `WM_SYSKEYDOWN` a un mensaje `WM_SYSCHAR` o `WM_SYSDEADCHAR`.

Generalmente ignoraremos todos estos mensajes, salvo `WM_CHAR`. Los mensajes `WM_SYSCHAR` y `WM_SYSDEADCHAR` se usan por Windows para acceder de forma rápida a menús, y no necesitamos procesarlos. En cuanto al mensaje `WM_DEADCHAR`, notifica sobre caracteres de teclas muertas, y generalmente, tampoco resultará interesante procesarlos.

## Teclas muertas

Veamos qué es este curioso concepto de tecla muerta. Las teclas muertas son aquellas que no generan un carácter por sí mismas, y necesitan combinarse con otras para formarlos. Por ejemplo, la tecla del acento (´), cuando se pulsa, no crea un carácter, es necesario pulsar otra tecla después para que eso ocurra. Si la tecla que se pulsa en segundo lugar genera un carácter que se puede combinar con la tecla muerta, se generará un único carácter, por ejemplo 'á'. Si no es así, se generan dos caracteres, el primero combinando la tecla muerta con un espacio, y el segundo con el carácter, por ejemplo "´b".

Cuando se pulse una tecla muerta, el mensaje que se genera por `TranslateMessage` puede ser `WM_DEADCHAR` o `WM_SYSDEADCHAR`, pero en cualquier caso, estos mensajes se puede ignorar, ya que el sistema los almacena internamente para generar los caracteres imprimibles.

```
case WM_CHAR:
    switch((TCHAR) wParam) {
        case 13:
```

```

        // Procesar retorno de línea
        break;
    case 0x08:
        // Procesar carácter de retroceso (borrar)
        break;
    default:
        // Cualquier otro carácter
        break;
}
InvalidateRect(hwnd, NULL, TRUE);
break;

```

## Estado de teclas

A veces nos interesa conocer el estado de alguna tecla concreto en el momento en que estamos procesando un mensaje procedente de otra pulsación de tecla. Por ejemplo, para tratar combinaciones de teclas como ALT+Fin o ALT+Inicio. Tenemos dos funciones para hacer esto.

Por una parte, la función [GetAsyncKeyState](#) nos dice el estado de una tecla virtual en el mismo momento en que la llamamos. Y la función [GetKeyState](#) nos da la misma información, pero en el momento en que se generó el mensaje que estamos tratando.

```

    case WM_KEYDOWN: // CONTROL+Inicio = Borra todo
        if(VK_HOME == (int)wParam) { // Tecla de inicio
            if(GetKeyState(VK_CONTROL) && 0x1000) {
                nLinea=0;
                nColumna=0;
                lista[0][0] = 0;
                InvalidateRect(hwnd, NULL, TRUE);
            }
        }
        break;

```

En este ejemplo, usamos la combinación CTRL+Inicio para borrar el texto que estamos escribiendo. Procesamos el mensaje [WM\\_KEYDOWN](#), para detectar la tecla de [Inicio], y si cuando eso

sucede, verificamos si también está pulsada la tecla de [CTRL], para ello usamos la función [GetKeyState](#) y comprobamos si el valor de retorno tiene el bit de mayor peso activo, comparando con 0x1000.

## Ejemplo 38

### Hot keys

He preferido no traducir el término "hot key", ya que me parece que es mucho más familiar que la traducción literal "tecla caliente". Una *hot key* es una tecla, o combinación de teclas, que tiene asignada una función especial y directa.

En Windows hay muchas hot keys predefinidas, por ejemplo, Ctrl+Alt+Supr sirve para activar el administrador de tareas, o la tecla de Windows izquierda en combinación con la tecla 'E', para abrir el explorador de archivos. Dentro de cada ventana o aplicación existen más, por ejemplo, Alt+F4 cierra la ventana, etc.

Hay dos tipos de hot keys, uno es el de las asociadas a ventanas. Es posible asociar una tecla o combinación de teclas a una ventana, de modo que al pulsarla se activa esa ventana, estemos donde estemos, estas son las hot keys globales.

El otro tipo, que es el que vamos a ver ahora, son las hot keys de proceso, lo locales. Nuestra aplicación puede crear tantas de ellas como creamos necesario, procesarlas y, si es necesario, destruirlas.

Crear, o mejor dicho, registrar una hot key es sencillo, basta con usar la función [RegisterHotKey](#). Esta función necesita cuatro parámetros. El primero es la ventana a la que estará asociada la hot key. El segundo parámetro es el identificador. El tercero son los modificadores de tecla, indica si deben estar presionadas las teclas de Control, Alt, Mayúsculas o Windows. Y el cuarto es el código de tecla virtual asociado a la hot key. Recordemos que los códigos de teclas virtuales de teclas correspondientes a caracteres son los propios caracteres, en el caso de letras, las mayúsculas.

```

case WM_CREATE:
    hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
    color = GetSysColor(COLOR_BACKGROUND);
    RegisterHotKey(hwnd, ID_VERDE, 0, 'V');
    RegisterHotKey(hwnd, ID_ROJO, MOD_ALT, 'R');
    RegisterHotKey(hwnd, ID_AZUL, MOD_CONTROL, 'A');
    break;

```

Cada vez que se pulse la tecla o combinación de teclas correspondiente a una hot key, el sistema la busca entre las registradas, y envía un mensaje [WM\\_HOTKEY](#) a la ventana que la registró. Aunque esa ventana no esté activa, el mensaje será enviado, siempre que sea la única ventana que ha registrar esa combinación de teclas. En el parámetro *wParam* se recibe el identificador de la hot key.

```

case WM_HOTKEY:
    switch((int)wParam) {
        case ID_VERDE:
            color = RGB(0,255,0);
            break;
        case ID_ROJO:
            color = RGB(255,0,0);
            break;
        case ID_AZUL:
            color = RGB(0,0,255);
            break;
    }
    InvalidateRect(hwnd, NULL, FALSE);
    break;

```

Finalmente, se puede desregistrar una hot key usando la función [UnregisterHotKey](#), indicando la ventana para la que se registró, y el identificador.

```

case WM_DESTROY:
    UnregisterHotKey(hwnd, ID_VERDE);

```

```

        UnregisterHotKey(hwnd, ID_ROJO);
        UnregisterHotKey(hwnd, ID_AZUL);
        PostQuitMessage(0);    /* envía un mensaje
WM_QUIT a la cola de mensajes */
        break;

```

## Ejemplo 39

### Códigos de teclas virtuales

Los códigos virtuales de las teclas que generan caracteres son los códigos ASCII de esos caracteres, por ejemplo, el código virtual de la tecla [A] es el 'A', o en número, el 65. Para el resto de las teclas existen constantes definidas en el fichero "winuser.h". Las constantes definidas son:

Constante	Tecla	Constante	Tecla
VK_LBUTTON	Botón izquierdo de ratón	VK_RBUTTON	Botón derecho de ratón
VK_CANCEL		VK_MBUTTON	Botón central de ratón
VK_BACK		VK_TAB	Tabulador
VK_CLEAR		VK_RETURN	Retorno
VK_KANA		VK_SHIFT	Mayúsculas
VK_CONTROL	Control	VK_MENU	
VK_PAUSE	Pausa	VK_CAPITAL	Bloqueo mayúsculas
VK_ESCAPE	Escape	VK_SPACE	Espacio
VK_PRIOR	Página anterior	VK_NEXT	Página siguiente
VK_END	Fin	VK_HOME	Inicio
VK_LEFT	Flecha izquierda	VK_UP	Flecha arriba
VK_RIGHT	Flecha	VK_DOWN	Flecha abajo



	derecha		
VK_SELECT		VK_PRINT	Imprimir pantalla
VK_EXECUTE		VK_SNAPSHOT	
VK_INSERT	Insertar	VK_DELETE	Suprimir
VK_HELP	Ayuda	VK_LWIN	Windows izquierda
VK_RWIN	Windows derecha	VK_APPS	Menú de aplicación
VK_NUMPAD0	'0' numérico	VK_NUMPAD1	'1' numérico
VK_NUMPAD2	'2' numérico	VK_NUMPAD3	'3' numérico
VK_NUMPAD4	'4' numérico	VK_NUMPAD5	'5' numérico
VK_NUMPAD6	'6' numérico	VK_NUMPAD7	'7' numérico
VK_NUMPAD8	'8' numérico	VK_NUMPAD9	'9' numérico
VK_MULTIPLY	Multiplicar	VK_ADD	Sumar
VK_SEPARATOR		VK_SUBTRACT	Restar
VK_DECIMAL	Punto decimal	VK_DIVIDE	Dividir
VK_F1	F1	VK_F2	F2
VK_F3	F3	VK_F4	F4
VK_F5	F5	VK_F6	F6
VK_F7	F7	VK_F8	F8
VK_F9	F9	VK_F10	F10
VK_F11	F11	VK_F12	F12
VK_F13	F13	VK_F14	F14
VK_F15	F15	VK_F16	F16
VK_F17	F17	VK_F18	F18
VK_F19	F19	VK_F20	F20

VK_F21	F21	VK_F22	F22
VK_F23	F23	VK_F24	F24
VK_NUMLOCK	Bloqueo numérico	VK_SCROLL	Bloqueo desplazamiento
VK_LSHIFT	Mayúsculas izquierdo	VK_RSHIFT	Mayúsculas derecho
VK_LCONTROL	Control izquierdo	VK_RCONTROL	Control derecho
VK_LMENU		VK_RMENU	
VK_PROCESSKEY		VK_ATTN	
VK_CRSEL		VK_EXSEL	
VK_EREOF		VK_PLAY	
VK_ZOOM		VK_NONAME	
VK_PA1		VK_OEM_CLEAR	

Las teclas sin descripción no están en mi teclado, de modo que no he podido averiguar a qué corresponden.

# Capítulo 35 Cadenas de caracteres

Windows trata las cadenas de caracteres de un modo algo distinto a como lo hacen las funciones estándar C. Esto se debe a que Windows maneja varios conjuntos de caracteres: ANSI, que son los que ya conocemos, como caracteres de ocho bits y Unicode, que son de dos bytes.

También puede manejar, diferentes formas de comparar y ordenar cadenas, diferentes configuraciones de idioma, que afectan a la forma de representar mayúsculas y minúsculas, o de comparar caracteres, etc. Por ejemplo, en español, la letra 'ñ' es mayor que la 'n' y menor que la 'o'. En inglés ni siquiera existe esa letra. Otro ejemplo, si intentamos obtener la mayúscula de la letra 'ñ' usando funciones estándar, el resultado no será la 'Ñ'.

## Recursos de cadenas

Al igual que podemos crear recursos para mapas de bits, menús, iconos, etc, también podemos crearlos para almacenar cadenas y leerlas desde la aplicación. Esto tiene varias ventajas y aplicaciones.

Los recursos de una aplicación pueden ser modificados por un editor adecuado sin modificar la parte ejecutable de una aplicación. Esto permite traducir una aplicación a distintos idiomas sin tener que compilar la aplicación ni tener que compartir el código fuente.

Es más, podemos crear nuestras aplicaciones para que sean multilenguaje, de modo que usen las cadenas adecuadas según la configuración de la aplicación.

## Fichero de recursos

Lo primero que debemos crear es una tabla de cadenas (stringtable) dentro del fichero de recursos, esto se hace mediante la sentencia **STRINGTABLE**:

```
STRINGTABLE
BEGIN
    ID_TITULO,      "Título de la aplicación"
    ID_SALUDO,      "Hola, estoy preparado para empezar."
    ID_DESPEDIDA,   "Gracias por usar esta aplicación."
END
```

Como se ve, una tabla de cadenas empieza con la sentencia **STRINGTABLE** y a continuación, entre un bloque **BEGIN-END** una lista de identificadores y cadenas entre comillas, separadas con una coma. En este caso, como suele ser nuestra costumbre, los identificadores son etiquetas definidas en nuestro fichero de cabecera de identificadores, aunque podría tratarse de números enteros de 16 bits.

El objetivo es hacer nuestra aplicación tan independiente del idioma como sea posible. Esto significa que en la aplicación no deberían aparecer cadenas literales, sino que deben usar cadenas de recurso. De este modo, sólo con traducir las cadenas del fichero de recursos, todos los literales usados en la aplicación cambiarán de idioma. Esto evita tener que repasar todos los ficheros fuente buscando literales para sustituirlos, y tener que compilar la aplicación de nuevo.

## Cargar cadenas desde recursos

Para obtener una cadena desde un recurso se usa la función **LoadString**. Esta función necesita cuatro parámetros. El primero es un manipulador de instancia, generalmente a la misma instancia de nuestra aplicación, aunque como veremos en capítulos más avanzados, también podemos obtener cadenas desde otros módulos, o desde DLLs. El segundo parámetro es el identificador de

la cadena, el tercero el búfer donde se recibe la cadena leída, y el cuarto el tamaño de dicho búfer.

Por ejemplo:

```
static HINSTANCE hInstance;
char mensaje[64];
char titulo[64];
...
case WM_CREATE:
    hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
    LoadString(hInstance, ID_TITULO, titulo, 64);
    LoadString(hInstance, ID_SALUDO, mensaje, 64);
    MessageBox(hwnd, mensaje, titulo, MB_OK);
    break;
```

## Funciones para cadenas

Algunas funciones estándar C tienen una versión repetida en el API de Windows. En el caso de las siguientes funciones, están mejoradas para manipular cadenas Unicode:

Windows	C estándar
<a href="#">lstrcat</a>	<a href="#">strcat</a>
<a href="#">lstrcmp</a>	<a href="#">strcmp</a>
<a href="#">lstrcmpi</a>	<a href="#">strcmpi</a>
<a href="#">lstrcpy</a>	<a href="#">strcpy</a>
<a href="#">lstrlen</a>	<a href="#">strlen</a>

Por ejemplo, la versión Windows de [strlen](#), [lstrlen](#) siempre calcula la longitud de una cadena en caracteres, independientemente de si los caracteres son de uno o dos bytes.

```
// Comparar cadenas:
if(lstrcmp("Niño", "Ñape") < 0)
    lstrcpy(mensaje, "Niño es menor que Ñape");
else
    lstrcpy(mensaje, "Niño es mayor que Ñape");
```

```
TextOut(hdc, 10, 280, mensaje, strlen(mensaje));  
if(lstrcmp("Ola", "Ñape") < 0)  
    lstrcpy(mensaje, "Ola es menor que Ñape");  
else  
    lstrcpy(mensaje, "Ola es mayor que Ñape");  
TextOut(hdc, 10, 300, mensaje, strlen(mensaje));
```

La salida de este fragmento de programa es la que cabría esperar:

```
Niño es menor que Ñape  
Ola es mayor que Ñape
```

Es decir, la función considera que la letra 'Ñ' está entre la 'N' y la 'O', como realmente ocurre en español. Siempre y cuando nuestro Windows esté instalado en español, o el usuario haya seleccionado ese idioma, claro.

Otros casos donde es necesario crear nuevas funciones es cuando necesitamos operar con caracteres dentro de cadenas. Como en Windows los caracteres pueden ser de uno o dos bytes, moverse a lo largo de una cadena puede no ser siempre tan directo como usando cadenas C estándar. Disponemos de dos funciones para movernos dentro de cadenas: [CharNext](#) para avanzar al siguiente carácter de una cadena, y [CharPrev](#) para retroceder al anterior.

Algo parecido pasa con la conversión de mayúsculas a minúsculas, y viceversa. En este caso disponemos de cuatro funciones:

Función	Descripción
<a href="#">CharLower</a>	Convertir un carácter o una cadena a minúsculas.
<a href="#">CharLowerBuff</a>	Convertir un carácter o una cadena a minúsculas.
<a href="#">CharUpper</a>	Convertir un carácter o una cadena a mayúsculas.

## CharUpperBuff Convertir un carácter o una cadena a mayúsculas.

```
static char alfabeto[] = "abcdefghijklmnñopqrstuvwxyz  
áéíóúëü ç";  
...  
        // Mayúsculas y minúsculas  
        CharLower(alfabeto);  
        TextOut(hdc, 10, 220, alfabeto,  
strlen(alfabeto));  
        CharUpper(alfabeto);  
        TextOut(hdc, 10, 240, alfabeto,  
strlen(alfabeto));
```

También en este caso el resultado es el esperado, las letras se convierten a mayúscula y minúscula correctamente, aunque se trate de caracteres con acentos, diéresis o tildes:

```
abcdefghijklmnñopqrstuvwxyz áéíóúëü ç  
ABCDEFGHIJKLMNÑOPQRSTUVWXYZ ÁÉÍÓÚËÜ Ç
```

Otro grupo de funciones estándar que se ven afectadas por el modo de trabajar en Windows son las del grupo de "ctype". En este caso tenemos otras cuatro funciones:

Función	Descripción
<a href="#">IsCharAlpha</a>	Verificar si un carácter es alfabético.
<a href="#">IsCharAlphaNumeric</a>	Verificar si un carácter es alfanumérico.
<a href="#">IsCharLower</a>	Verifica si un carácter está en minúscula.
<a href="#">IsCharUpper</a>	Verifica si un carácter está en mayúscula.

Por último, tenemos un par de funciones que sustituyen a las funciones estándar **sprintf** y **vsprintf**. Se trata de **wsprintf** y

wvsprintf.

```
for(i = 0; i < 10; i++) {  
    wsprintf(mensaje, "Cadena formateada %d: valor  
%c",  
            i+1, alfabeto[i]);  
    TextOut(hdc, 10, i*20, mensaje,  
strlen(mensaje));  
}
```

En Windows debemos usar las variantes del API, ya que están preparadas para trabajar con cadenas y caracteres Unicode, algo que las funciones estándar no pueden hacer.

Hay que tener en cuenta que estas funciones no aceptan las mismas cadenas de formato que **sprintf** o **vsprintf**, por ejemplo, no sirven para valores en punto flotante o punteros. Sin embargo, tienen más opciones para cadenas Unicode/ANSI.

## Ejemplo 40



# Capítulo 36 Aceleradores

Los aceleradores son atajos para los menús. Lo normal y deseable es que nuestras aplicaciones proporcionen aceleradores para las opciones más frecuentes de los menús.

Un acelerador es una pulsación de tecla, o de teclas, que producen el mismo efecto que una selección en un menú. Windows detecta la pulsación y convierte el mensaje de teclado a un mensaje [WM\\_COMMAND](#) o [WM\\_SYSCOMMAND](#).

Cuando el usuario se familiariza con los aceleradores de teclado puede ahorrar mucho tiempo al activar comandos, ya que es mucho más rápido pulsar una tecla que activar una opción de menú, ya sea mediante el teclado o el ratón.

## Recursos de aceleradores

Como ya hemos visto con los menús, cuadros de diálogo, cadenas, etc, en el caso de los aceleradores también podemos crearlos como un recurso, y cargarlos por la aplicación cuando se necesiten.

### Fichero de recursos

Para crear una tabla de aceleradores se usa la sentencia [ACCELERATORS](#):

```
aceleradores ACCELERATORS
BEGIN
    VK_F1, CM_OPCION1, VIRTKEY /* F1 */
    "^C", CM_SALIR /* Control C */
    "K", CM_OPCION2 /* K */
    "k", CM_OPCION3, ALT /* Alt k */
END
```

```

    0x34, CM_OPCION4, ASCII    /* 4 */
    VK_F2, CM_OPCION5, ALT, SHIFT, VIRTKEY    /* Alt Mays F2
*/
    "1", CM_OPCION6, ALT, CONTROL, VIRTKEY    /* Alt Control 1
*/
END

```

Si queremos que nuestros aceleradores no distingan mayúsculas de minúsculas, es mucho mejor definirlos a partir de teclas virtuales.

## Cargar aceleradores desde recursos

Para cargar los aceleradores desde un recurso se usa la función [LoadAccelerators](#). Como ya viene siendo corriente con este grupo de funciones, esta también necesita dos parámetros. El primero es un manipulador de la instancia del módulo que contiene el recurso, el segundo, es el identificador de recurso.

```

HACCEL hAcelerador = LoadAccelerators(hThisInstance,
"aceleradores");

```

## Bucle de mensajes para usar aceleradores

Para que nuestra aplicación reciba mensajes cuando se pulsan las teclas que definen los aceleradores hay que modificar el bucle de mensajes. Los mensajes [WM\\_KEYDOWN](#) y [WM\\_SYSKEYDOWN](#) deben traducirse a mensajes [WM\\_COMMAND](#) y [WM\\_SYSCOMMAND](#), y para ello hay que usar la función [TranslateAccelerator](#):

```

while(TRUE == GetMessage(&mensaje, NULL, 0, 0))
{
    /* Traducir mensajes de teclas a mensajes de
acelerador */
    if(!TranslateAccelerator(hwnd, hAcelerador,

```

```
&mensaje)) {  
    /* Traducir mensajes de teclas virtuales a  
    mensajes de caracteres  
        sólo si TranslateAccelerator regresa con nulo  
    */  
    TranslateMessage(&mensaje);  
    /* Enviar mensaje al procedimiento de ventana */  
    DispatchMessage(&mensaje);  
}  
}
```

## Crear tablas de aceleradores sin usar recursos

Se usa la función [CreateAcceleratorTable](#) para crear tablas de aceleradores durante la ejecución, a partir de un array de estructuras [ACCEL](#). El manipulador de aceleradores obtenido mediante esta función se puede usar igual que el obtenido por la función [LoadAccelerators](#).

Un detalle importante: las tablas de aceleradores creadas mediante [CreateAcceleratorTable](#) se deben destruir antes de terminar la aplicación mediante una llamada a la función [DestroyAcceleratorTable](#). Esto no es necesario cuando se usa la función [LoadAccelerators](#).

En general usaremos aceleradores de recursos, ya que son más fáciles de manejar. Sin embargo, este procedimiento nos permitiría, por ejemplo, crear aceleradores definidos por el usuario.

## Combinar aceleradores y menús

En general, los aceleradores estarán ligados a opciones de menú, pero no hay nada que informe al usuario sobre los aceleradores disponibles, salvo que nosotros lo indiquemos directamente.

Una forma fácil de hacerlo es añadir la información del acelerador al ítem del menú. Seguro que has notado que algunas opciones de menú tienen una combinación de teclas a su derecha, por ejemplo, en el menú de sistema de cualquier ventana la última opción es la de "Cerrar", y a su derecha aparece el texto "Alt+F4". Eso es un acelerador.

Nosotros podemos hacer lo mismo con nuestros menús. Es muy fácil añadir información a la derecha del texto de un ítem, basta con insertar la secuencia "\a", y a continuación el texto del acelerador. El texto después de la secuencia "\a" se justifica a la derecha:

```
menu MENU
BEGIN
  POPUP "&Principal"
  BEGIN
    MENUITEM "Opción &1\aF1", CM_OPCION1
    MENUITEM "Opción &2\aK", CM_OPCION2
    MENUITEM "Opción &3\aAlt+k", CM_OPCION3
    MENUITEM "Opción &4\a4", CM_OPCION4
    MENUITEM "Opción &5\aAlt+Mays+F2", CM_OPCION5
    MENUITEM "Opción &6\aAlt+Ctrl+1", CM_OPCION6
    MENUITEM SEPARATOR
    MENUITEM "&Salir\a^C", CM_SALIR
  END
END
```

## Aceleradores globales

Existen varios aceleradores definidos a nivel global de Windows, nuestras aplicaciones deben intentar evitar definir esos aceleradores, aunque en principio no es imposible hacerlo, sencillamente no es aconsejable. Los aceleradores son:

Acelerador	Descripción
ALT+ESC	Cambia a la siguiente aplicación.
ALT+F4	Cierra una aplicación o ventana.
ALT+HYPHEN	Abre el menú de sistema de una

	ventana de documento.
ALT+PRINT SCREEN	Copia una imagen de la ventana activa al portapapeles.
ALT+SPACEBAR	Abre el menú de sistema para una ventana de aplicación.
ALT+TAB	Cambia a la siguiente aplicación.
CTRL+ESC	Cambia a la lista de tareas de Windows (menú de Inicio).
CTRL+F4	Cierra el grupo activo o ventana de documento.
F1	Arranca la ayuda si la aplicación la tiene.
PRINT SCREEN	Copia una imagen de la pantalla al portapapeles.
SHIFT+ALT+TAB	Cambia a la aplicación anterior. El usuario debe presionar Alt+Mays mientras presiona TAB.

## Diferencia entre acelerador y menú

Usar un acelerador es prácticamente lo mismo que activar un ítem de un menú. En ambos casos se envía un mensaje `WM_COMMAND` o `WM_SYSCOMMAND`, y nuestro procedimiento de ventana lo procesará del mismo modo. En un caso el identificador será el que hemos usado para el acelerador, y en el otro el que hemos usado para el ítem del menú.

Sin embargo es posible que a veces nos interese saber si un comando procede de un acelerador o de un menú. Para saberlo podemos comprobar la presencia de un bit, el código de notificación del mensaje `WM_COMMAND`, que se proporciona en la palabra de mayor peso del parámetro *wParam*.

## Ejemplo 41

# Capítulo 37 Menús 2

En el capítulo 5 tratamos el tema de los menús, pero de una manera superficial. La intención era dar unas nociones básicas para poder usar menús en nuestros primeros ejemplos. Ahora los veremos con más detalle, y estudiaremos muchas características que hasta ahora habíamos pasado por alto.

Los menús pueden tener, por ejemplo, mapas de bits a la izquierda del texto. También pueden comportarse como checkboxes o radiobuttons. Se pueden inhibir ítems. Podemos crear menús flotantes contextuales al pulsar con el ratón sobre determinadas zonas de la aplicación, etc.

## Marcas en menús

Seguro que estás familiarizado con las marcas de chequeo que aparecen en algunos ítems de menú en casi todas las aplicaciones Windows. Es frecuente que se puedan activar o desactivar opciones, y que se pueda ver el valor actual de cada opción consultando menú. El funcionamiento es exactamente el mismo que el de los checkboxes y radiobuttons.

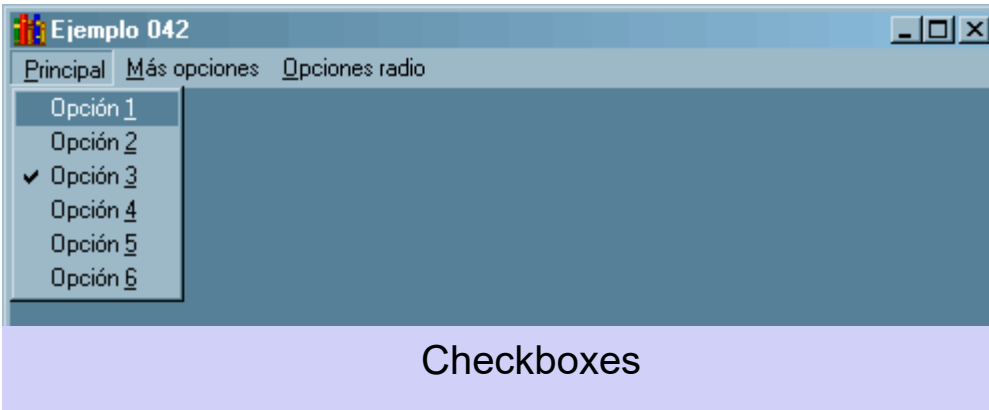
Hemos visto que los ítems de menú se comportan exactamente igual que los botones, pero hasta ahora sólo hemos usado los menús como un conjunto de "Pushbuttons", veremos qué otras opciones tenemos.

## Menús como checkboxes

Recordemos que los checkboxes son uno o un conjunto de botones, cada uno de los cuales puede tener dos estados. Cada uno de los botones dentro de un conjunto de checkboxes puede estar

marcado o no. De hecho, cada checkbox se comporta de un modo independiente, y sólo se agrupan conceptualmente, es decir, los grupos no son controlados por Windows.

Con menús podemos crear este efecto para cada ítem, añadiendo un mapa de bits a la izquierda que indique si la opción está marcada o no. Generalmente, cuando no lo esté, se eliminará la marca.



Disponemos de dos funciones para marcar ítems de menú. La más

sencilla de usar es [CheckMenuItem](#). Esta función necesita tres parámetros: el primero es el manipulador del menú, el segundo el identificador o la posición que ocupa el ítem, y el tercero dos banderas que indican si el segundo parámetro es un identificador o una posición y si el ítem se va a marcar o no.

Entonces, lo primero que necesitamos, es una forma de obtener un manipulador del menú de la ventana. Esto es sencillo: usaremos la función [GetMenu](#), que nos devuelve el manipulador del menú asociado a una ventana.

Lo segundo será decidir si accedemos al ítem mediante un identificador o mediante su posición. La primera opción es la mejor, ya que usando el identificador no necesitamos un manipulador al submenú concreto que tiene nuestro ítem. Eso siempre que los identificadores no estén duplicados, en ese caso necesitamos usar la segunda opción.

Otra cosa que necesitamos es averiguar si un ítem está o no marcado. Para ello podemos usar la función [GetMenuState](#), que usa prácticamente los mismos parámetros que [CheckMenuItem](#), salvo

que el tercero sólo indica si el segundo es un identificador o una posición.

Por ejemplo, este sencillo código averigua si un ítem está o no marcado, y cambia el estado de la marca:

```
if (GetMenuState (GetMenu (hwnd), CM_OPCION1, MF_BYCOMMAND)
& MF_CHECKED)
    CheckMenuItem (GetMenu (hwnd), CM_OPCION1, MF_BYCOMMAND
| MF_UNCHECKED);
else
    CheckMenuItem (GetMenu (hwnd), CM_OPCION1, MF_BYCOMMAND
| MF_CHECKED);
```

Sin embargo, la documentación del API de Windows dice que la función [CheckMenuItem](#) es obsoleta, aunque se puede seguir usando, y recomienda usar en su lugar la función [SetMenuItemInfo](#). Esta función permite modificar otros valores, como veremos a lo largo de este capítulo.

Al tener más opciones, esta función es más complicada de usar. Necesita cuatro parámetros: el manipulador de menú, el identificador o posición del ítem, un tercer parámetro que indica si el segundo es un identificador o una posición y un puntero a una estructura [MENUITEMINFO](#).

Esta estructura contiene campos que indican qué valores queremos modificar, y otros campos para indicar los nuevos valores. En nuestro caso, queremos modificar el valor de chequeo, por lo tanto, asignaremos el valor [MIIM\\_STATE](#) al campo *fMask* y al campo *fState* el valor apropiado: [MFS\\_CHECKED](#) o [MFS\\_UNCHECKED](#).

La misma estructura se usa para recuperar valores de un ítem de menú, pero con la función [GetMenuItemInfo](#), el valor del campo *fState* nos dirá si el ítem está o no marcado.

```
MENUITEMINFO infoMenu;
...
infoMenu.cbSize = sizeof (MENUITEMINFO);
```



```

infoMenu.fMask = MIIM_STATE;
GetMenuItemInfo(GetMenu(hwnd), CM_OPCIONA, FALSE,
&infoMenu);
if(infoMenu.fState & MFS_CHECKED)
    infoMenu.fState = MFS_UNCHECKED;
else
    infoMenu.fState = MFS_CHECKED;
SetMenuItemInfo(GetMenu(hwnd), CM_OPCIONA, FALSE,
&infoMenu);

```

Por supuesto, a lo largo del programa siempre podremos consultar el estado de estos ítems, de modo que sabremos qué opción ha activado el usuario cuando lo necesitemos. Además, los ítems siguen generando mensajes [WM\\_COMMAND](#), así que podemos procesarlos cuando sean modificados, en lugar de usarlos como simples editores de opciones.

## Menús como radiobuttons

Cuando tenemos un grupo de opciones de las que sólo una de ellas puede estar activa, estamos ante un conjunto de radiobuttons. Estos botones sí deben estar agrupados, y se necesitan al menos dos de ellos en un grupo. En este caso, Windows sí puede gestionar el grupo automáticamente, para asegurarse que al marcar una opción, la que estaba activa anteriormente se desactive.



En este caso, procesar estos ítems es más simple, una única

función bastará para gestionar todo un grupo de radioítems: [CheckMenuItem](#). Como primer parámetro tenemos el

manipulador de menú, el segundo es el identificador o posición del primer ítem del grupo, el tercero el del último ítem del grupo, el cuarto el del ítem a marcar y el quinto indica si los parámetros segundo a cuarto son identificadores o posiciones.

Este ejemplo marca la opción 3 dentro de un grupo de 1 a 6. Automáticamente elimina la marca del ítem que la tuviese previamente:

```
CheckMenuItem(GetMenu(hwnd),CM_RADIO1, CM_RADIO6,  
CM_RADIO3, MF_BYCOMMAND);
```

Al usar esta función, la marca normal (V) se sustituye por la del círculo negro.

Podemos seguir usando las funciones [GetMenuState](#) o mejor, [GetMenuItemInfo](#), para averiguar qué ítem es el activo en un momento dado.

## Ejemplo 42

### Inhibir y oscurecer ítems

También frecuente que en determinadas circunstancias queramos que algunas opciones no estén disponibles para el usuario, ya sea porque no tienen sentido, o por otra razón. Por ejemplo, esto es lo que pasa con la opción de "Maximizar" del menú de sistema cuando la ventana está maximizada.



En  
ese  
sentido,  
los ítems  
pueden  
tener tres  
estados

distintos: activo, inhibido y oscurecido. Hasta ahora sólo hemos trabajado con ítems activos. Los inhibidos tienen el mismo aspecto para el usuario, pero no se pueden seleccionar. Los oscurecidos además de no poderse seleccionar, aparecen en gris o difuminados, para indicar que están inactivos.

Podemos cambiar el estado de acceso de un ítem usando la función [EnableMenuItem](#), o mejor, con la función [SetMenuItemInfo](#). Aunque la documentación del API dice que la primera está obsoleta, se puede seguir usando si no se necesitan otras características de la segunda.

La función [EnableMenuItem](#) necesita tres parámetros, el primero es el manipulador de menú, el segundo su identificador o posición, y el tercero indica si el segundo es un identificador o una posición y el estado que se va a asignar al ítem.

En este ejemplo se usa la función [EnableMenuItem](#) para inhibir y oscurecer un ítem, y la función [SetMenuItemInfo](#) para activarlo, de modo que se ilustran los dos modos de realizar esta tarea.

```
MENUITEMINFO infoMenu;
...
switch(LOWORD(wParam)) {
    case CM_INHIBIR:
        EnableMenuItem(GetMenu(hwnd), CM_OPCION, MF_DISABLED | MF_BYCOMMAND);
        CheckMenuRadioItem(GetMenu(hwnd), CM_INHIBIR, CM_ACTIVAR, CM_INHIBIR, MF_BYCOMMAND);
        break;
    case CM_OSCURECER:
        EnableMenuItem(GetMenu(hwnd), CM_OPCION, MF_GRAYED | MF_BYCOMMAND);
        CheckMenuRadioItem(GetMenu(hwnd), CM_INHIBIR, CM_ACTIVAR, CM_OSCURECER, MF_BYCOMMAND);
        break;
    case CM_ACTIVAR:
        infoMenu.cbSize = sizeof(MENUITEMINFO);
        infoMenu.fMask = MIIM_STATE;
        infoMenu.fState = MFS_ENABLED;
        SetMenuItemInfo(GetMenu(hwnd), CM_OPCION, FALSE, &infoMenu);
}
```

```
CheckMenuItem(GetMenu(hwnd), CM_INHIBIR,  
CM_ACTIVAR, CM_ACTIVAR, MF_BYCOMMAND);  
break;
```

Lo mismo se puede hacer con [ModifyMenu](#), aunque esta función es obsoleta y se desaconseja su uso.

## Ejemplo 43

## Más sobre ficheros de recursos

También en lo que respecta a los ficheros de recursos hay más cosas que contar. Para empezar, además de la sentencia [MENU](#) existe otra sentencia para crear recursos de menús extendidos, que incorporan características introducidas en Windows 95. Pero de todos modos, aún no hemos visto todo sobre la sentencia [MENUITEM](#).

### Sentencia MENUITEM y POPUP

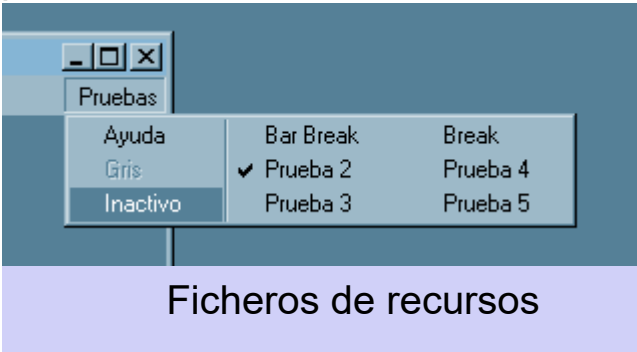
La sintaxis de [MENUITEM](#) es:

```
MENUITEM texto, resultado, [lista_de_opciones]
```

Y la de [POPUP](#) es:

```
POPUP texto, [lista_de_opciones]  
BEGIN  
    definiciones_de_item  
    ...  
END
```

Hasta ahora nunca hemos usado las opciones (que evidentemente son opcionales). Ahora que hemos visto las marcas y los estados de los ítems de menú, veremos que esas opciones pueden ser interesantes:



- **GRAYED**: el nombre del ítem esta inicialmente inactivo y aparece en el menú en gris o ligeramente degradado. En el ejemplo, el ítem "Gris".
- **HELP** identifica un ítem de

ayuda. Esto hace que se muestre alineado a la derecha. En el ejemplo, todo el **POPUP** "Pruebas" ha sido definido con esta opción.

- **INACTIVE** el nombre del ítem se muestra, pero no puede ser seleccionado. En el ejemplo, el ítem "Inactivo".
- **MENUBREAK** coloca el ítem de menú en una nueva línea en ítems de menú. En menús pop-up, coloca el ítem de menú en una nueva columna sin división entre las columnas. En el ejemplo es el caso del ítem "Break".
- **MENUBARBREAK** lo mismo que **MENUBREAK** excepto que para menús pop-up, separa la nueva columna de la anterior con una línea vertical. En el ejemplo, "Bar break".
- **CHECKED**: el ítem tiene una marca de verificación junto a él. En el ejemplo, el ítem "Prueba 2" está definido con esta opción. Esta opción sólo se aplica a ítems de menú, no a popups.

```
menu MENU
BEGIN
  POPUP "&Pruebas"
  BEGIN
    MENUITEM "&Ayuda", 500
    MENUITEM "&Gris", 501, GRAYED
    MENUITEM "&Inactivo", 502, INACTIVE
    MENUITEM "Ba&r Break", 503, MENUBARBREAK
    MENUITEM "Prueba &2", 504, CHECKED
    MENUITEM "Prueba &3", 505
```

```
MENUITEM "&Break", 506, MENUBREAK
MENUITEM "Prueba &4", 507
MENUITEM "Prueba &5", 508
END
END
```

## Detalles sobre cadenas de ítems

En cuanto a las cadenas de caracteres que se usan en los ítems, hay varias secuencias de caracteres que nos permiten cierto control sobre el aspecto de las cadenas.

Ya hemos visto que se puede seleccionar una de las letras para que se pueda activar el ítem mediante el teclado, basta como poner el carácter & delante de la letra elegida.

Si queremos que el carácter & aparezca como parte del texto del ítem, habrá que duplicar el carácter &&, por ejemplo "&Carácter &&" aparecerá como **Carácter &**.

Del mismo modo, para incluir comillas dobles dentro de la cadena, se deben escapar mediante la secuencia \", por ejemplo "Cadena \"prueba\"" aparecerá como **Cadena "prueba"**.

Por último, podemos usar la secuencia \a para justificar el texto a la derecha, y la secuencia \t para introducir un tabulador. Estas secuencias se suelen usar para añadir la información sobre las teclas aceleradoras de menú, como vimos en el capítulo anterior.

## Sentencia MENUEX

Como habrás visto, en las sentencias anteriores no es posible crear menús con todas las características que hemos explicado. Por ejemplo, no se pueden crear ítems con la marca de "radio" en lugar de la de "check". Los menús evolucionaron a partir de Windows 95, y para adaptarse a ello, se creó una nueva sentencia: **MENUEX**.

La sintaxis es la siguiente:

```

menuID MENUEX
BEGIN
    [[MENUITEM itemText [, [id] [, [type [| state]]]] |
    [POPUP itemText [, [id] [, [type [| state] [, helpID]]]]
    BEGIN
        popupBody
    END]] ...]
END

```

En realidad, la sintaxis de **MENUEX** es la misma que la de **MENU**, lo que cambia es que las sentencias **MENUITEM** y **POPUP** tienen más posibilidades dentro de una sentencia **MENUEX**.

Como vemos, en el caso de **MENUITEM**, tanto el identificador, como el tipo, como es estado son opcionales.

Además, tanto para el tipo como para el estado, existen más posibilidades que en el **MENUITEM** anterior, en concreto podemos usar las constantes de tipo definidas en el fichero "winuser.h" para la estructura **MENUITEMINFO**, que empiezan con MFT\_. Para eso hay que incluir ese fichero de cabecera en el fichero de recursos.

Estas constantes son:

- **MFT\_BITMAP**: muestra el ítem de menú usando un mapa de bits. En realidad, esta constante no se puede usar en ficheros de recursos, y se debe modificar en el fichero ejecutable. Veremos esto más abajo.
- **MFT\_MENUBARBREAK**: coloca el ítem de menú en una línea nueva (para una barra de menú) o en una columna nueva (para un menú desplegable, un submenú o un menú de atajo). Para este último caso, una línea vertical separa la nueva columna de la antigua. Equivale a **MENUBARBREAK** de **MENU**.
- **MFT\_MENUBREAK**: coloca el ítem de menú en una línea nueva (para una barra de menú) o en una columna nueva (para un menú desplegable, un submenú o un menú de atajo). Para este último caso, la columna no se separa con una línea vertical. Equivale a **MENUBREAK**.

- **MFT\_OWNERDRAW**: asigna la responsabilidad del trazado del ítem de menú a la ventana a la que pertenece el menú. Esta bandera tampoco se puede usar en un fichero de recursos.
- **MFT\_RADIOCHECK**: muestra los ítems marcados usando la marca del radio-button en lugar una marca de chequeo si el miembro `hbmChecked` es **NULL**. Esta bandera tampoco se puede usar en un fichero de recursos.
- **MFT\_RIGHTJUSTIFY**: justifica a la derecha el ítem de menú y los ítems siguientes. Este valor sólo es válido si el ítem de menú está en una barra de menú. Equivale a **HELP**.
- **MFT\_SEPARATOR**: especifica que el ítem de menú es un separador. Un ítem de menú separador aparece como una línea horizontal divisora. No es válido en barras de menú. Equivale al **SEPARATOR** de **MENU**. Tampoco se puede usar en ficheros de recursos, para poner separadores basta con usar una cadena nula, `""`.
- **MFT\_STRING**: muestra el ítem de menú usando una cadena de texto. Es el valor por defecto, no es necesario especificarlo.

Para el estado podemos usar las constantes declaradas en "winuser.h" con el prefijo **MFS\_**.

- **MFS\_CHECKED**: marca el ítem de menú. Equivale a **CHECKED**.
- **MFS\_DEFAULT**: especifica que el ítem de menú es el ítem por defecto. Esta bandera no se puede usar en un archivo de recursos. Veremos esto más abajo.
- **MFS\_DISABLED**: inhibe el ítem de menú de modo que no puede ser seleccionado, pero no lo oscurece. Equivale a **INACTIVE**.
- **MFS\_ENABLED**: desinhibe el ítem de menú de modo que pueda ser seleccionado. Este es el estado por defecto, y no es necesario especificarlo.
- **MFS\_GRAYED**: inhibe el ítem de menú y lo oscurece de modo que no puede ser seleccionado. Equivale a **GRAYED**.
- **MFS\_HILITE**: resalta el ítem de menú. Esta bandera tampoco se puede usar en ficheros de recursos.



- **MFS\_UNCHECKED**: quita la marca del ítem de menú. Este es el estado por defecto, tampoco es necesario especificarlo.
- **MFS\_UNHILITE**: elimina el resaltado del ítem de menú. Este es el estado por defecto, tampoco es necesario especificarlo.

```
menu MENUEX
BEGIN
  POPUP "&Principal"
  BEGIN
    MENUITEM "&Inhibir",    CM_INHIBIR, MFT_STRING
    MENUITEM "&Oscurecer", CM_OSCURECER, MFT_STRING
    MENUITEM "&Activar",    CM_ACTIVAR, MFT_STRING
    MENUITEM "" // MFT_SEPARATOR
    MENUITEM "O&pción",      CM_OPCION, MFT_STRING
  END
  POPUP "&Pruebas",0,MFT_STRING | MFT_RIGHTJUSTIFY
  BEGIN
    MENUITEM "&Ayuda", 500, MFT_STRING
    MENUITEM "&Gris",  501, MFT_STRING, MFS_GRAYED
    MENUITEM "&Inactivo", 502, MFT_STRING, MFS_DISABLED
    MENUITEM "Ba&r Break", 503, MFT_STRING | MFT_MENUBARBREAK
    MENUITEM "Prueba &2", 504, MFT_STRING, MFS_CHECKED
    MENUITEM "Prueba &3", 505, MFT_STRING, MFS_UNCHECKED
    MENUITEM "&Break", 506, MFT_STRING | MFT_MENUBREAK
    MENUITEM "Prueba &4", 507, MFT_STRING
    MENUITEM "Prueba &5", 508, MFT_STRING
  END
END
```

#### **Nota:**

He encontrado tres errores en la documentación de Windows.

1. Las definiciones de las constantes **MFS\_DISABLED** y **MFS\_GRAYED**, dentro del fichero "winuser.h" actualmente están modificadas. La primera macro debe valer 2, y la segunda 1; actualmente ambas valen 3.

2. En la documentación original de **MENUEX** dice que los valores de tipo y estado deben estar separados con una coma. He comprobado que en algunas versiones antiguas del compilador de recursos, para que todo funcione adecuadamente, estos valores se deben combinar usando el operador de bits OR (|) o el de suma (+).

3. Los separadores dentro de menús **POPUP** no se consiguen con la bandera de tipo **MFS\_SEPARATOR**, sino usando una cadena nula ("").

Resumamos un poco:

## Items marcados y no marcados

Bueno, ya hemos visto que cuando definimos recursos de menú podemos elegir el estado inicial de las marcas de chequeo. En el caso de **MENU** mediante los modificadores *CHECKED* para marcado, y nada para no marcados. En el caso de **MENUEX** mediante las banderas **MFS\_CHECKED** y **MFS\_UNCHECKED**.

Ya hemos visto que no podemos hacer nada para que los ítems sean de tipo radio en el fichero de recursos. Esta característica se modifica durante la ejecución.

## Items activos, inactivos u oscurecidos

Por defecto, los ítems de menú estarán activos, pero si queremos crearlos inactivos u oscurecidos, en el caso del recurso **MENU** usaremos los modificadores *INACTIVE* o *GRAYED* respectivamente. En el caso del recurso **MENUEX** las banderas **MFS\_DISABLED** para inhibido, o **MFS\_GRAYED** para oscurecido.

## Separadores y líneas de ruptura

Existen varias opciones para separar ítems, ya sea horizontalmente o verticalmente, y en barras de menú o en menús desplegables.

En barras de menú, y con recursos `MENU` podemos usar los modificadores `MENUBREAK` o `MENUBARBREAK` para cambiar de línea un ítem o un popup. También podemos usar el modificar `HELP` para trasladar un ítem o popup a la derecha de la barra de menú.

En el caso del recurso `MENUEX`, el mismo efecto se consigue con las banderas `MFT_MENUBREAK`, `MFT_MENUBARBREAK` o `MFT_RIGHTJUSTIFY`, respectivamente.

Dentro de los menús popup, los modificadores del recurso `MENU` tienen efectos diferentes. El modificador `MENUBREAK` hace que el siguiente ítem se sitúe en una nueva columna, y el modificador `MENUBARBREAK` lo mismo, pero se añade una línea vertical separadora.

Los separadores horizontales, en el caso del recurso `MENU` se consiguen con una línea `MENUITEM SEPARATOR`, estos separadores son útiles para agrupar ítems, o grupos de opciones del tipo `checkitems` o `radioitems`.

En el caso de recursos `MENUEX` los separadores verticales se consiguen con las banderas `MFT_MENUBREAK` y `MFT_MENUBARBREAK`, y los horizontales con una cadena vacía `MENUITEM ""`.

**Nota:**

`SEPARATOR` es probablemente una etiqueta para la cadena `""`, y son intercambiables.

## Cargar recursos

Para cargar un recurso de menú tenemos varias opciones, como ya vimos en el capítulo 5. Lo más simple es usar un menú de clase,

asignando la cadena con el nombre del menú al miembro *lpzMenuName* de la estructura [WNDCLASSEX](#) o [WNDCLASS](#), por ejemplo:

```
wincl.lpszMenuName = "menu";
```

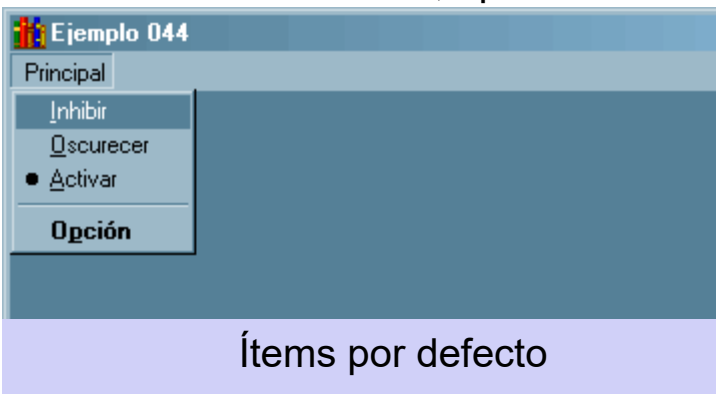
También podemos cargar recursos de menú y usar un manipulador para referirnos a ellos durante la ejecución, mediante la función [LoadMenu](#):

```
HMENU hMenu;  
...  
hMenu = LoadMenu(hThisInstance, "menu");
```

Este menú se puede asignar a una ventana mediante la función [SetMenu](#) o al crear la ventana con las funciones [CreateWindow](#) o [CreateWindowEx](#), pero más abajo veremos que existen otras posibilidades.

## Ítems por defecto

Para cada popup se puede marcar un ítem para que sea el ítem por defecto. Cuando hagamos doble clic sobre el popup, será como si hubiésemos seleccionado el ítem por defecto. Para indicar al usuario cual es ese ítem, aparecerá marcado en negrita.



El ítem por defecto hay que marcarlo durante la ejecución, ya hemos visto que no se puede hacer en el fichero de recursos. Para hacerlo

---

disponemos de la función

### [SetMenuDefaultItem](#).

Pero esta función hay que manejarla con cuidado, ya que en cada menú desplegable sólo puede haber un ítem por defecto, como primer parámetro deberemos pasar un manipulador del submenú donde se encuentre el ítem. Como además, no tiene mucho sentido marcar un ítem por defecto en una barra de menú (ya que siempre será accesible mediante un clic, y un doble clic sobre una barra de menú no hace nada), generalmente deberemos usar la función [GetSubMenu](#) para obtener un manipulador al submenú que contiene el ítem a marcar por defecto. Esto se debe hacer aunque el identificador del ítem a marcar sea único:

```
SetMenuDefaultItem(GetSubMenu(GetMenu(hwnd), 0),  
CM_OPCION, FALSE);
```

Este ejemplo marcará el ítem con el identificador CM\_OPCION en el primer submenú del menú de la ventana *hwnd*.

Para obtener el ítem por defecto de un submenú podemos usar la función [GetMenuDefaultItem](#), con las mismas precauciones:

```
id = GetMenuDefaultItem(GetSubMenu(GetMenu(hwnd), 0),  
FALSE, GMDI_GOINTOPOPUPS);
```

El primer parámetro es un manipulador de submenú, el segundo indica el tipo de valor devuelto: **FALSE** para que devuelva un identificador, y **TRUE** para que devuelva una posición. El tercero indica el modo de realizar la búsqueda, el valor usado en el ejemplo busca recursivamente.

También podemos usar las funciones [SetMenuItemInfo](#) y [GetMenuItemInfo](#) para hacer las mismas tareas.

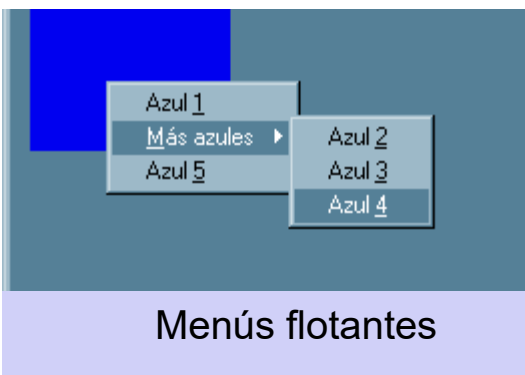
En concreto, usar la función [SetMenuItemInfo](#) es más práctico, ya que no necesitamos obtener un manipulador de submenú, y

marcará el ítem por defecto independientemente de cual sea el submenú donde esté:

```
infoMenu.cbSize = sizeof(MENUITEMINFO);  
infoMenu.fMask = MIIM_STATE  
infoMenu.fState = MFS_DEFAULT;  
SetMenuItemInfo(GetMenu(hwnd), CM_OPCION, FALSE,  
&infoMenu);
```

## Ejemplo 44

### Menús flotantes o contextuales



Otra posibilidad de los menús es crear menús flotantes, también llamados menús de atajo o menús contextuales. Estos menús se suelen mostrar cuando el usuario hace clic con el ratón sobre distintas zonas de la ventana, y normalmente se muestran distintos menús

dependiendo de la zona, o mejor dicho, del contexto.

Todo esto es responsabilidad del programador: procesar las pulsaciones de botones de ratón, decidir qué menú mostrar, y finalmente mostrarlo en pantalla.

La aplicación recibe el mensaje [WM\\_CONTEXTMENU](#) cuando el usuario hace clic sobre la ventana, aunque también podemos procesar los mensajes de ratón comunes.

Para mostrar el menú flotante en pantalla se usa la función [TrackPopupMenuEx](#):

```
HMENU hmenu;  
...
```

```

    case WM_CREATE:
        hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
        hmenu = LoadMenu(hInstance, "menu2");
    ...
    case WM_CONTEXTMENU:
        TrackPopupMenuEx(GetSubMenu(hmenu, 0),
            TPM_CENTERALIGN | TPM_HORIZONTAL | TPM_RIGHTBUTTON,
            LOWORD(lParam), HIWORD(lParam),
            hwnd, NULL);
        break;
    ...
    case WM_DESTROY:
        DestroyMenu(hmenu);
    ...

```

El primer parámetro es un manipulador del menú popup que queremos visualizar, debe ser un manipulador de menú creado mediante la función [CreatePopupMenu](#) o, como en este ejemplo, extraído de un menú existente, que a su vez puede ser uno asignado a una ventana, o cargado de un recurso.

Hay que recordar que hay que destruir los menús que no estén asignados a una ventana mediante la función [DestroyMenu](#).

En el ejemplo 45, que ilustra el uso de los menús flotantes, usamos un menú de recurso para crear todos los posibles menús flotantes, y elegimos el que nos interesa en cada caso mediante la función [GetSubMenu](#).

El segundo parámetro son banderas que nos ayudan a situar el menú con relación a las coordenadas que indicamos en los parámetros tercero y cuarto.

El quinto parámetro identifica la ventana que recibirá los mensajes generados por el menú, y el sexto, nos permite definir un área de exclusión, donde no se mostrará el menú, si es posible.

## Ejemplo 45

### Acceso por teclado

Ya comentamos en el capítulo dedicado al ratón que ese dispositivo no es imprescindible en Windows, de modo que la comunicación entre la aplicación y el usuario se puede hacer exclusivamente desde el teclado, y los menús no son una excepción.

## **Mnemónicos**

Una de las formas de agilizar el acceso por teclado son los mnemónicos, que se asignan añadiendo el símbolo & delante de la letra que activa el menú popup o el ítem de menú.

## **Acceso de teclado estándar**

Además existen teclas y combinaciones de teclas especialmente dedicadas al acceso a los menús:

- **Mnemónico:** el carácter correspondiente selecciona el primer ítem con ese mnemónico. Si el ítem abre un menú, éste se muestra y se resalta su primer ítem. En caso contrario, se selecciona ese ítem. Para activar un menú mediante el mnemónico es necesario pulsar la tecla ALT.
- **ALT:** enciende y apaga la barra de menú.
- **ALT+SPACEBAR:** muestra el menú de ventana (o menú de sistema).
- **ENTER:** activa un menú y selecciona su primer ítem de menú, si es que ese ítem abre un submenú. En caso contrario, se selecciona el ítem como si el usuario hubiese soltado el botón del ratón mientras el ítem estaba seleccionado.
- **ESC:** sale del modo de menú.
- **Flecha izquierda:** regresa al ítem de menú previo. Los ítems de mayor nivel incluyen los nombres y el menú de sistema. Si el ítem seleccionado está en una barra de menú, se selecciona la columna anterior de la columna o el menú de mayor nivel previo.



- Flecha derecha: funciona como la tecla de flecha izquierda, pero en dirección contraria. En barras de menú, esta tecla mueve hacia la derecha una columna; cuando el ítem seleccionado actualmente es la última de la derecha abre, se selecciona el siguiente menú.
- Flechas arriba y abajo: activa un menú cuando se presiona en un nombre de menú. Cuando se presiona en un menú, la flecha arriba selecciona el ítem previo y la flecha abajo el siguiente.

## Aceleradores

Otro modo de acceder a opciones de menú mediante el teclado son los aceleradores, aunque en realidad estos son independientes de los menú, es costumbre añadir a los menús las combinaciones de teclas que activan la misma opción mediante un acelerador. De este modo, el usuario puede memorizar más fácilmente aquellas opciones de menú que usa más frecuentemente, y acceder a ellas mediante aceleradores en lugar de hacerlo mediante el menú.

## Modificar menús

Siempre es posible modificar un menú durante la ejecución de la aplicación, tan sólo necesitamos su manipulador y aplicar las funciones que necesitemos entre las siguientes:

Función	Descripción
<a href="#">AppendMenu</a>	Añade un nuevo ítem de menú al final del menú especificado. Esta función ha sido sustituida por <a href="#">InsertMenuitem</a> . Sin embargo, se puede seguir usando, si no se necesitan las características extendidas de la función <a href="#">InsertMenuitem</a> .
<a href="#">InsertMenu</a>	Inserta un nuevo ítem de menú dentro de un menú, moviendo los

otros ítems hacia abajo. Esta función también ha sido sustituida por la función [InsertMenuItem](#). De todos modos, se puede seguir usando, si no se necesitan las características extendidas de [InsertMenuItem](#).

[InsertMenuItem](#) Inserta un nuevo ítem de menú en la posición especificada de un menú. Usa una estructura [MENUITEMINFO](#) para crear el ítem.

[ModifyMenu](#) Modifica un ítem de menú existente. Esta función ha sido sustituida por [SetMenuItemInfo](#). De todos modos, se puede seguir usando si no se necesitan las características extendidas de [SetMenuItemInfo](#).

[SetMenuItemInfo](#) Modifica la información sobre un ítem de menú. Usa una estructura [MENUITEMINFO](#) para modificar el ítem.

[DeleteMenu](#) Borra un ítem del menú especificado. Si el ítem de menú abre un menú o submenú, esta función destruye el manipulador del menú o submenú y libera la memoria usada por él.

Si se modifica un menú que actualmente se está visualizando, se debe llamar a la función [DrawMenuBar](#) para actualizar la ventana y reflejar los cambios.

## El menú de sistema

El menú de sistema, también llamado menú de **ventana** o menú de control, es el que se muestra cuando se hace clic sobre el icono de la aplicación, o cuando se pulsa Alt+espacio.

Cuando se crea una aplicación, Windows asigna siempre el menú de sistema por defecto, siempre que se especifique el estilo `WS_SYSMENU` al crear la ventana.

A diferencia de los menús que hemos usado hasta ahora, el menú de sistema genera mensajes `WM_SYSCOMMAND`, en lugar de mensajes `WM_COMMAND`, y generalmente, salvo que modifiquemos el menú de sistema, dejaremos que el proceso por defecto se encargue de esos mensajes, mediante la función `DefWindowProc`.

## Modificar el menú de sistema

Modificar el menú de sistema es relativamente simple. Para empezar, obtendremos un manipulador de menú del menú de sistema mediante la función `GetSystemMenu`, usaremos como parámetros el manipulador de la ventana, y el valor `FALSE`, de modo que obtenemos una copia modificable del menú de sistema actual.

Una vez disponemos de un manipulador de menú para una copia del menú de sistema, podemos modificarlo usando las funciones que ya conocemos: `AppendMenu`, `InsertMenu`, `InsertMenuItem`, `SetMenuItemInfo`, `ModifyMenu` o `DeleteMenu`.

Los identificadores de los ítems que insertemos deben ser menores que 0xf000.

```
static HMENU hmenuSistema;
...
switch(msg) { /* manipulador del mensaje */
    case WM_CREATE:
        hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
        hmenuSistema = GetSystemMenu(hwnd, FALSE);
        InsertMenu(hmenuSistema, 0, MF_BYPOSITION |
```

```
MF_STRING | MF_POPUP,  
        (UINT) GetSubMenu(GetMenu(hwnd), 1), "&Opciones");  
        break;  
    ...
```

Los ítems incluidos en un menú de sistema generan mensajes **WM\_SYSCOMMAND**, por lo tanto, deberemos procesar este mensaje, teniendo cuidado de remitir a la función **DefWindowProc** aquellos que no procesemos nosotros.

```
case WM_SYSCOMMAND:  
    switch(LOWORD(wParam)) {  
        case CM_OPCION1:  
            break;  
        case CM_OPCION2:  
            break;  
        case CM_OPCION3:  
            break;  
        case CM_OPCION4:  
            break;  
        default:  
            return DefWindowProc(hwnd, msg, wParam,  
lParam);  
    }  
    break;
```

## Ejemplo 46

### Destrucción de menús

Los menús asociados a ventanas se destruyen automáticamente cuando se destruyen las ventanas, de modo que sólo es necesario destruir los menús que no estén asociados a ninguna ventana, en general, serán los que usemos como menús flotantes, o como menús alternativos, por ejemplo, si disponemos de diferentes versiones de menús para nuestras ventanas, y los usamos alternativamente, en función de las circunstancias.

Para destruir uno de esos menús se usa la función [DestroyMenu](#).

## Mensajes de menú

Ya hemos visto algunos de los mensajes que generan los menús, en concreto los que generan cuando se seleccionan ítems:

Función	Descripción
<a href="#">WM_COMMAND</a>	Es enviado cuando el usuario selecciona un comando de un ítem de un menú.
<a href="#">WM_SYSCOMMAND</a>	Cuando el usuario elige un comando desde el menú de <b>ventana</b> .

Pero hay más, algunos de los más interesantes pueden ser:

Función	Descripción
<a href="#">WM_INITMENU</a>	Se envía cuando un menú se va a activar. Eso ocurre cuando el usuario hace clic sobre un ítem de la barra de menú o cuando presiona la tecla de menú. Nos permite modificar el menú antes de que se muestre.
<a href="#">WM_INITMENUPOPUP</a>	Se envía cuando un menú emergente o un submenú va a ser activado. Esto permite a una aplicación modificar el menú antes de que sea mostrado, sin modificar el menú completo.
<a href="#">WM_MENUSELECT</a>	Se envía a la ventana cuando el usuario selecciona un ítem del menú.
<a href="#">WM_CONTEXTMENU</a>	Notifica a una ventana que

el usuario ha hecho clic con el botón derecho del ratón en la ventana.

## Mapas de bits en ítems de menú

hay varios modos de incluir mapas de bits en ítems de menú, en este capítulo veremos dos de ellos, dejaremos el método owner-draw para capítulos posteriores.

### Modificar mapas de bits de check

Una de las formas de insertar mapas de bits, sin modificar el resto del ítem, es cambiar las marcas de chequeo por defecto por las diseñadas por nosotros. Esto se consigue con la función [SetMenuItemBitmaps](#) o con [SetMenuItemInfo](#).

```
static HBITMAP hSi;
static HBITMAP hNo;
...
    hSi = LoadBitmap(hInstance,
"si");
    hNo = LoadBitmap(hInstance,
"no");

    infoMenu.cbSize =
sizeof(MENUIITEMINFO);
    infoMenu.fMask = MIIM_CHECKMARKS;
    infoMenu.hbmpChecked = hSi;
    infoMenu.hbmpUnchecked = hNo;
    for(i = 0; i < 3; i++)
        SetMenuItemInfo(GetSubMenu(GetMenu(hwnd), 0),
            i, TRUE, &infoMenu);
...
DeleteObject(hNo);
DeleteObject(hSi);
```



Lo mismo, usando la función [SetMenuItemBitmaps](#):

```

...
hSi = LoadBitmap(hInstance, "si");
hNo = LoadBitmap(hInstance, "no");
for(i = 0; i < 3; i++)
    SetMenuItemBitmaps(GetSubMenu(GetMenu(hwnd), 0),
        i, MF_BYPOSITION, hSi, hNo);
...

```

Primero deberemos diseñar los mapas de bits del tamaño adecuado para ambas marcas, o al menos para la marca de check, ya que la correspondiente al ítem no marcado puede dejarse en blanco. Para conseguir el tamaño de los mapas de bits se puede usar la función [GetMenuCheckMarkDimensions](#).

```

wsprintf(mensaje, "%d %d",
    HIWORD(GetMenuCheckMarkDimensions()),
    LOWORD(GetMenuCheckMarkDimensions()));
MessageBox(hwnd, mensaje, "Medidas de mapas de bits",
    MB_OK);

```

## Items de mapas de bits

También se pueden usar mapas de bits en lugar de texto en los ítems de menú. Para hacerlo hay que crear o modificar los ítems en la ejecución del programa, ya que no es posible hacerlo al crear el fichero de recursos.



Como siempre, disponemos de dos formas de hacerlo, mediante las funciones [AppendMenu](#), [InsertMenu](#) y [ModifyMenu](#), o mediante [InsertMenuItem](#) y [SetMenuItemInfo](#), y la estructura [MENUITEMINFO](#).

Recordemos el proceso para crear un mapa de bits durante la

---

ejecución:

1. Usar la función [CreateCompatibleDC](#) para crear un contexto de dispositivo compatible con el usado por la ventana principal de la aplicación.
2. Usar la función [CreateCompatibleBitmap](#) para crear un mapa de bits compatible con la ventana principal de la aplicación o usar la función [CreateBitmap](#) para crear un mapa de bits monocromo.
3. Usar la función [SelectObject](#) para seleccionar el mapa de bits en el contexto de dispositivo compatible.
4. Usar las funciones de trazado del GDI, como [Ellipse](#), [LineTo](#) o [BitBlt](#), para generar una imagen dentro del mapa de bits.

En el primer caso, insertaremos o modificaremos los ítems indicando que son de tipo [MF\\_BITMAP](#), e indicando como nuevo ítem un manipulador de mapa de bits.

```
InsertMenu(hmenu, 0, MF_BYPOSITION | MF_BITMAP,  
           CM_OPCION, (LPCTSTR)(hBitmap));
```

En el segundo caso, insertaremos o modificaremos los ítems de tipo [MFT\\_BITMAP](#), e indicando un manipulador de mapa de bits.

```
infoMenu.cbSize = sizeof(MENUITEMINFO);  
infoMenu.fMask = MIIM_TYPE | MIIM_ID;  
infoMenu.fType = MFT_BITMAP;  
infoMenu.wID = CM_OPCION;  
infoMenu.dwTypeData = (LPSTR)(hBitmap);  
InsertMenuItem(hmenu, 0, TRUE, &infoMenu);
```

## Ejemplo 47



# Capítulo 38 La memoria

Windows tiene su manera particular de manejar la memoria del sistema. Esto es lógico, y era de esperar, ya que como recurso que es, la memoria también debe ser controlada y administrada por el sistema operativo.

## Memoria virtual

Para administrar la memoria, en el API32, Windows mantiene un espacio *virtual* de memoria con direcciones de 32 bits para cada proceso. Esto permite que cada uno de los procesos disponga de hasta cuatro gigabytes de memoria. Dos de esos gigas están disponibles para el usuario, los correspondientes a las primeras direcciones de memoria; los otros dos están reservados para el núcleo del sistema.

Se trata de un espacio *virtual*, esto quiere decir que las aplicaciones no acceden directamente a memoria, y que las direcciones que manejamos no corresponden a direcciones de memoria física. El sistema se encarga de mapear esas direcciones virtuales a direcciones físicas. En esto, como en todos los recursos, el sistema operativo trabaja como intermediario entre el usuario y el hardware.

Este modo de trabajar proporciona a cada aplicación gran cantidad de memoria, de hecho, en la mayoría de los casos proporciona más memoria de la disponible físicamente. Para mantener toda esa memoria se trabaja con un fichero en disco como almacén de memoria complementaria: **el fichero de paginación o fichero de intercambio**.

El sistema de memoria *virtual* trabaja con unidades de memoria llamadas *páginas*. El tamaño de cada página varía dependiendo del

tipo de ordenador, (en microprocesadores de la familia x86 suele ser de 4 KB). Mientras es posible, las páginas de memoria se asignan a cada proceso desde la memoria física, pero cuando es necesaria más memoria, los procesos inactivos pueden copiar algunas de sus páginas en el fichero de intercambio y liberar de ese modo parte de la memoria física que se puede asignar al proceso activo.

Todo esto es transparente para el usuario, que sólo notará que cuando el sistema está muy cargado aumenta la actividad del disco y disminuye la velocidad del sistema.

Nos interesa saber, de todos modos, que cada página de memoria virtual asociada a un proceso puede estar en uno de tres posibles estados:

- *Libre*: no es accesible, pero puede ser reservada o asignada.
- *Reservada*: no se usa por el proceso, ni tampoco está asociada a memoria física ni ocupa espacio en el fichero de intercambio. El proceso tampoco podrá obtener memoria de páginas reservadas.
- *Asignada*: página que tiene memoria física o espacio en el fichero de intercambio asociado. Estas páginas pueden ser protegidas, para evitar su acceso o limitarlo a sólo lectura, o pueden ser accedidas en lectura y escritura.

**Nota:**

Puedes ver más detalles sobre la memoria virtual en el artículo de José Manuel: [Memoria virtual](#).

## Un poco de historia

Vamos a ver algunos conceptos que tal vez te suenen, pero que en el API de Win32 han quedado obsoletos. En cualquier caso,

puede ser conveniente saber algo sobre ellos, recordar sus aplicaciones y comprender por qué ya no son necesarios.

## Memoria local y global

En las primeras versiones de Windows, de 16 bits, existen dos tipos de memoria que se pueden usar para crear objetos de memoria para un proceso: la memoria *local* y la memoria *global*.

Esto tiene su fundamento en el modo de trabajar de los procesadores de **Intel**. En el modo real la memoria se divide en segmentos, en cada uno de los cuales se usan punteros de 16 bits para acceder a la memoria, debido a esto, cada segmento tiene un tamaño de 64KB. Estos punteros de 16 bits son llamados punteros *near* o cercanos. Para acceder a la memoria fuera del segmento se usan punteros *far* o lejanos, que son punteros de 32 bits.

Hasta Windows 3.1 se trabajaba en modo real, y a cada proceso se le asignaba un segmento. La memoria que se puede obtener por el proceso dentro de su propio segmento se denomina **memoria local**, y la que se obtiene fuera de él, se denomina **memoria global**. Los bloques de memoria local tienen que ser, por definición, pequeños, ya que su tamaño máximo estaba limitado al espacio libre en el segmento.

A partir de Windows 95 y en Windows NT, es decir, con el API de 32 bits, el procesador trabaja en modo protegido. El concepto de segmento desaparece, y la memoria es lineal, con direcciones de 32 bits, lo que proporciona acceso a 4GB de memoria. Además se introduce el modelo de memoria *virtual*, lo cual elimina cualquier diferencia entre memoria *local* y *global*, y en el API32 las funciones para reservar y liberar memoria *local* o *global* son equivalentes, y siempre retornan punteros de 32 bits, o sea, punteros *far*.

## Otros atributos de la memoria en Windows

Además de lo comentado hasta ahora, Windows mantiene otros dos atributos para cada objeto de memoria obtenido: la movilidad y la descartabilidad.

## Objetos móviles y fijos

El primer atributo se refiere a la movilidad del objeto. De nuevo estamos ante un concepto que ha perdido sentido en el API32. En versiones de Windows de 16 bits cada objeto podría tener el atributo de movilidad activado o desactivado.

¿Por qué crear objetos móviles? Debido al modo en que se organiza la memoria en Windows de 16 bits, después de un tiempo de funcionamiento del sistema, la memoria podía estar muy fragmentada, con pequeños bloques de memoria reservados para distintos objetos, y pequeños huecos resultantes de la destrucción de objetos innecesarios. Esto puede provocar que, a pesar de existir suficiente memoria libre en el sistema, sea imposible conseguir un bloque del tamaño necesario para satisfacer una nueva petición.

La solución es que el sistema pueda trasladar algunos de los objetos existentes a otras posiciones, de modo que se desfragmente la memoria y las partes libres queden contiguas.

Pero esto crea un conflicto, ya que Windows es un sistema multitarea, habrá aplicaciones que estén usando ciertos objetos, de modo que tales objetos no pueden ser movidos. O bien, aunque esos objetos no se estén usando, los punteros que los manejan tienen valores constantes, o deben conservar sus valores mientras el proceso que los ha creado siga funcionando. De otro modo sería imposible manejarlos y liberarlos.

Cuando se crea un objeto de memoria fijo el sistema proporciona un puntero de 32 bits, y se pueden manejar como punteros corrientes:

1. **Creación:** se crea un objeto fijo de memoria, y se obtiene un puntero.
2. **Uso:** podemos trabajar con él ya que el valor del puntero es fijo.

3. **Destrucción:** cuando ya no sea necesario destruimos el objeto, liberando la memoria asociada.

Sin embargo, los objetos de memoria móviles creados usando el API de Windows no proporcionan una dirección fija, sino un manipulador de memoria. El proceso es el siguiente:

1. **Creación:** se crea un objeto movable de memoria, y se obtiene un manipulador.
2. **Bloqueo:** cuando se va a usar el objeto se bloquea el objeto, y se obtiene una dirección para la memoria correspondiente.
3. **Uso:** mientras permanece bloqueado, el objeto no será movido por el sistema, y podemos trabajar con él como si el valor del puntero fuese fijo.
4. **Desbloqueo:** cuando no necesitamos manipular el objeto, lo desbloqueamos, y el sistema podrá moverlo si lo considera necesario.
5. **Destrucción:** cuando ya no sea necesario, y si no está bloqueado, destruimos el objeto, liberando la memoria asociada y el manipulador.

Por supuesto, crear objetos móviles tiene la ventaja de que el sistema puede gestionar la memoria de un modo mucho más eficaz, pero siempre es posible, si nuestro programa lo requiere, crear objetos fijos. La diferencia es que en este caso, el sistema no podrá mover tales objetos para desfragmentar la memoria.

## **Objetos descartables y no descartables**

El otro atributo está relacionado con el mismo problema.

Cuando se debe desfragmentar la memoria para conseguir bloques libres lo suficientemente largos, es necesario mover el contenido de cada objeto, de modo que los valores que contienen no se pierdan. El concepto de memoria descartable va un paso más allá. Si el contenido de cierto objeto móvil es fácilmente recuperable o se puede reconstruir, sin necesidad de almacenarlo de forma

permanente, el sistema de gestión de memoria no necesita almacenarlo permanentemente, y puede mantener sólo los manipuladores, sin necesidad de mantener el contenido de tales objetos.

Por ejemplo, tenemos un mapa de bits en memoria creado a partir de un recurso. El contenido de la memoria correspondiente a ese mapa de bits puede ser recuperado del recurso tantas veces como sea necesario, pero, siempre que sea posible, nos conviene mantenerlo en memoria, ya que recuperar ese recurso requiere cierto tiempo. Si creamos ese objeto como descartable, el sistema puede borrar la memoria asociada cuando lo considere necesario, haciendo innecesario tanto mantener esa memoria como copiarla.

El proceso, cuando se trabaja con objetos descartables es algo más complicado:

1. **Creación:** se crea un objeto descartable de memoria, y se obtiene un manipulador.
2. **Bloqueo:** cuando se va a usar el objeto se bloquea el objeto, y se obtiene una dirección para la memoria correspondiente.
3. **Restitución:** se averigua si la memoria del objeto ha sido descartado por el sistema, y en ese caso se restituye su valor, ya sea por cálculo o por carga desde un fichero.
4. **Uso:** mientras permanece bloqueado, el objeto no será movido por el sistema, y podemos trabajar con él como si el valor del puntero fuese fijo.
5. **Desbloqueo:** cuando no necesitamos manipular el objeto, lo desbloqueamos, y el sistema podrá moverlo o descartarlo si lo considera necesario.
6. **Destrucción:** cuando ya no sea necesario destruimos el objeto, liberando la memoria asociada y el manipulador.

Cada vez que bloqueemos el objeto hay que verificar si su memoria ha sido descartada o no. Un objeto cuya memoria haya sido descartada contendrá basura.

Tampoco es posible crear objetos fijos descartables. Los objetos descartables siempre deben tener el atributo de movilidad.

## Funciones *clásicas* para manejo de memoria

Disponemos de nueve funciones para manejar memoria local y global. Los nombres son los mismos, cambiando el prefijo "Local" y "Global".

A modo de resumen, ya que no será frecuente que usemos estas funciones, podemos considerar esta tabla:

	<b>Local</b>	<b>Global</b>
<b>Reservar memoria</b>	<a href="#">LocalAlloc</a>	<a href="#">GlobalAlloc</a>
<b>Descartar objeto de memoria</b>	<a href="#">LocalDiscard</a>	<a href="#">GlobalDiscard</a>
<b>Información sobre objeto de memoria</b>	<a href="#">LocalFlags</a>	<a href="#">GlobalFlags</a>
<b>Liberar objeto de memoria</b>	<a href="#">LocalFree</a>	<a href="#">GlobalFree</a>
<b>Obtener un manipulador de objeto</b>	<a href="#">LocalHandle</a>	<a href="#">GlobalHandle</a>
<b>Bloquear objeto</b>	<a href="#">LocalLock</a>	<a href="#">GlobalLock</a>
<b>Reubicar objeto de memoria</b>	<a href="#">LocalRealloc</a>	<a href="#">GlobalRealloc</a>
<b>Tamaño de un objeto</b>	<a href="#">LocalSize</a>	<a href="#">GlobalSize</a>
<b>Desbloquear un objeto</b>	<a href="#">LocalUnlock</a>	<a href="#">GlobalUnlock</a>

Como complemento disponemos de la función [GlobalMemoryStatus](#) para obtener información sobre la memoria disponible en el sistema. Esta función usa una estructura [MEMORYSTATUS](#) para almacenar la información.

Ejemplo

Existen además, otras funciones útiles para manejar memoria:

- [CopyMemory](#): copiar bloques de memoria.
- [MoveMemory](#): mueve un bloque de memoria.
- [FillMemory](#): llenar bloque con un valor determinado.
- [ZeroMemory](#): llenar un bloque con ceros.

## Desventajas de este modelo de memoria

Afortunadamente, esto pertenece al pasado, salvo que tengas que crear aplicaciones para versiones de Windows de 16 bits, claro.

El modelo de memoria virtual hace que todo este tema de memoria móvil carezca de sentido, ya que nuestras aplicaciones no trabajan con memoria *real*, el sistema siempre puede mover cualquier página de memoria para liberar recursos, sin que eso afecte en modo alguno a las direcciones de memoria virtuales.

La memoria descartable aún puede resultar útil, ya que permite liberar recursos que se usan poco o que pueden regenerarse fácilmente.

Otra desventaja es que las funciones estándar para manejar memoria: `malloc` y `free` no funcionan de forma segura en el modelo de memoria del API de 16 bits, algo que sí sucede en el modelo *virtual*. En el API de 16 bits, la función `malloc` no puede obtener objetos de memoria móviles o descartables. Lo mismo sucede con los operadores `new` y `delete`.

Sólo en el caso en que queramos crear objetos descartables tendremos que hacer uso de las funciones del API para manejar memoria.

## Funciones para manejo de memoria virtual

Ya hemos comentado que gracias al uso del modelo de memoria virtual, salvo para crear objetos de memoria descartables, con el API32 podremos usar las funciones estándar C, o los operadores de C++ para manejar memoria dinámicamente. Sin embargo, el modelo de memoria virtual nos permite un control sobre la memoria que no está disponible si sólo usamos funciones y operadores estándar.

El modelo virtual nos permite hacer cosas como:

### Reservar direcciones de memoria virtual



Las funciones [VirtualAlloc](#) y [VirtualAllocEx](#) permiten reservar un rango de direcciones, sin asignarles memoria física, o asignándosela.

- Podemos reservar direcciones de memoria, sin asignarles memoria física, de modo que se deje espacio para que estructuras dinámicas de datos crezcan durante la ejecución.
- Asignar memoria a direcciones previamente reservadas.
- Hacer ambas cosas a la vez.

```
int *puntero;
...
/* Reservar memoria sin asignar almacenamiento físico */
puntero = VirtualAlloc(NULL, 100 * sizeof(int),
MEM_RESERVE, PAGE_NOACCESS);
/* Acomodar físicamente memoria previamente reservada */
VirtualAlloc(puntero, 100*sizeof(int), MEM_COMMIT,
PAGE_READWRITE);
/* Reservar y acomodar espacio para memoria de una vez */
puntero = VirtualAlloc(NULL, 100 * sizeof(int),
MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
```

## Liberar direcciones de memoria virtual

Las funciones [VirtualFree](#) y [VirtualFreeEx](#) permite realizar las operaciones inversas a las anteriores.

- Liberar direcciones de memoria reservadas.
- Liberar memoria asignada sin liberar las direcciones.
- Liberar direcciones reservadas y memoria asignada.

```
int *puntero;

/* Libera memoria */
VirtualFree(puntero, 0, MEM_RELEASE);
/* Desasigna memoria, la memoria sigue reservada */
VirtualFree(puntero, 100*sizeof(int), MEM_DECOMMIT);
```

---

## Bloquear páginas de memoria asignada

La función [VirtualLock](#) permite bloquear determinadas páginas para que siempre permanezcan en memoria física, y no puedan ser transferidas al fichero de paginación.

Para desbloquear estas páginas se usa la función [VirtualUnlock](#).

```
int *puntero;

...
VirtualLock(puntero, 100*sizeof(int));
...
VirtualUnlock(puntero, 100*sizeof(int));
```

## Establecer atributos de protección de acceso

Las funciones [VirtualProtect](#) y [VirtualProtectEx](#) permiten asignar atributos de lectura/escritura, sólo lectura o sin acceso.

```
int *puntero;
DWORD protant; /* Valor anterior de la protección */

VirtualProtect(puntero, 100*sizeof(int), PAGE_READONLY,
&protant);
```

## Obtener información sobre páginas de memoria

Las funciones [VirtualQuery](#) y [VirtualQueryEx](#) se usan para obtener información sobre rangos de páginas de memoria virtual.

Estas funciones obtienen datos sobre rangos de páginas de memoria virtual a través de una estructura [MEMORY\\_BASIC\\_INFORMATION](#). De este modo podemos saber el tamaño de un bloque de memoria, su estado, su tipo de protección,

etc. Esta estructura también nos informa de esos mismos valores en la primera llamada a [VirtualAlloc](#), independientemente del estado en que esté actualmente.

```
int *puntero;  
MEMORY_BASIC_INFORMATION mbi;  
  
VirtualQuery(puntero, &mbi,  
sizeof(MEMORY_BASIC_INFORMATION));
```

Generalmente, en nuestras aplicaciones, no necesitaremos recurrir a funciones del API para manejar memoria, ya que el sistema se encarga de gestionar las llamadas a las funciones estándar de memoria: [malloc](#), [free](#), etc, y el uso de los operadores `new` y `delete`, de modo que en realidad siempre usaremos memoria virtual. El sistema se encarga también de proporcionar el almacenamiento físico para esa memoria, y nosotros no tendremos que preocuparnos por esos temas.

Pero a veces puede ser necesario gestionar la memoria de nuestra aplicación, asegurarse de que va a existir memoria disponible en fases siguientes del programa, aunque no la vayamos a necesitar de forma inmediata, proteger ciertas zonas, etc. En esos casos será útil saber que existen estos mecanismos, y saber aplicarlos de forma adecuada a cada caso.

## Ejemplo 48

# Capítulo 39 Control Edit avanzado

En este capítulo, y en los siguientes vamos a comentar con más detalles los controles básicos que ya hemos visto previamente. Veremos algunas características más avanzadas de cada uno, los mensajes de notificación, y los mensajes que aún no conocemos de cada uno.

Empezaremos por el control edit, y veremos todo lo que no se explicó en el [capítulo 7](#).

## Insertar controles edit durante la ejecución

En general, el sistema que vamos a comentar puede ser aplicado a cualquier control, aunque veremos ejemplos para cada caso.

En realidad, un control no es otra cosa que una ventana, en el caso del control edit, se trata de una ventana de la clase "EDIT". Como ventana que es tiene su propio procedimiento de ventana, y por supuesto, se pueden crear esta clase de ventanas usando las funciones [CreateWindow](#) y [CreateWindowEx](#).

Este sistema nos permite insertar controles en cualquier ventana o cuadro de diálogo, en lugar de usar siempre cuadros de diálogo y ficheros de recursos para insertar los controles. Cuando las aplicaciones sean sencillas será frecuente que nos baste una ventana para realizar todas las entradas y salidas.

Por supuesto, la posibilidad de insertar controles durante la ejecución nos proporciona más flexibilidad, y mayor control sobre la aplicación en ciertas circunstancias, por ejemplo, evitar que los

cuadros de diálogo se puedan editar usando herramientas de edición de recursos.

Veamos un ejemplo de cómo insertar un control de edición en una ventana:

```
    HWND hctrl;
...
    case WM_CREATE:
        hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
        /* Insertar control Edit */
        hctrl = CreateWindowEx(
            0,
            "EDIT",          /* Nombre de la clase */
            "",              /* Texto del título, no tiene
*/
            ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_TABSTOP, /* Estilo */
            36, 20,          /* Posición */
            120, 20,         /* Tamaño */
            hwnd,            /* Ventana padre */
            (HMENU)ID_TEXTO, /* Identificador del control
*/
            hInstance,       /* Instancia */
            NULL);          /* Sin datos de creación de
ventana */
        /* Inicialización de los datos de la aplicación
*/
        SetDlgItemText(hwnd, ID_TEXTO, "Inicial");
        SetFocus(hctrl);
        return 0;
```

Como vemos, usamos los mismos valores que en el fichero de recursos: identificador, clase de ventana (en este caso "EDIT"), estilo, posición y dimensiones.

El identificador del control se suministra a través del parámetro hMenu, por lo que será necesario hacer un casting del identificador al tipo [HMENU](#).

Ahora será nuestro procedimiento de ventana el encargado de procesar los mensajes procedentes del control. Recordemos que en

los ejemplos que hemos visto hasta ahora esto lo hacía el procedimiento de diálogo.

## Cambiar la fuente de un control edit

Veremos a continuación algunas formas de personalizar los controles edit.

Al insertar los controles edit en un cuadro de diálogo, usando el fichero de recursos, podemos especificar la fuente que queremos usar, pero al insertarlo directamente mediante la función [CreateWindow](#) o [CreateWindowEx](#), Windows siempre usa la fuente del sistema por defecto.

Esto afecta al aspecto estético de nuestros controles, (la verdad es que la fuente del sistema es bastante fea), y una de las ventajas de las aplicaciones gráficas es precisamente poder elegir el aspecto que queremos que tengan.

Pero hay una solución, es posible modificar la fuente de un control edit enviando un mensaje [WM\\_SETFONT](#). El lugar apropiado es, por supuesto, al procesar el mensaje [WM\\_INITDIALOG](#), cuando se trate de cuadros de diálogo, o al procesar el mensaje [WM\\_CREATE](#), cuando se trate de ventanas.

En el parámetro `wParam` pasamos un manipulador de fuente, y usaremos la macro [MAKELPARAM](#) para crear un valor [LPARAM](#), en el que especificaremos la opción de repintar el control, que se almacena en la palabra de menor peso de [LPARAM](#).

Esto nos permite modificar la fuente durante la ejecución, reflejando los cambios en pantalla.

```
static HFONT hfont;
...
hfont = CreateFont(24, 0, 0, 0, 300,
    FALSE, FALSE, FALSE, DEFAULT_CHARSET,
    OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
    PROOF_QUALITY, DEFAULT_PITCH | FF_ROMAN,
    "Times New Roman");
```



```

*/
{
    case WM_CREATE:
        hcrtl = CreateWindowEx(...);
        pincel = CreateSolidBrush(RGB(0,255,0));
        SetFocus(hcrtl);
        return 0;
    case WM_CTLCOLOREDIT:
        SetBkColor((HDC)wParam, RGB(0,255,0));
        SetTextColor((HDC)wParam, RGB(255,255,255));
        return (LRESULT)pincel;
    case WM_DESTROY:
        DeleteObject(pincel);
        PostQuitMessage(0);    /* envía un mensaje
WM_QUIT a la cola de mensajes */
        break;
    ...
}

```

## Ejemplo 49

### Controles edit de sólo lectura

Uno de los estilos que se pueden aplicar a un control edit es el de sólo lectura. **ES\_READONLY**. Cuando se activa estilo el contenido del control no podrá ser modificado por el usuario.

Esto es, aparentemente, una contradicción. Bien pensado, un control edit cuyo contenido no puede ser modificado es un control estático. Sin embargo, en determinadas circunstancias puede que no sea tan absurdo, sobre todo si tenemos en cuenta que este estilo se puede modificar durante la ejecución. Esto puede ser útil si en ciertas situaciones, determinados valores están predefinidos. Por ejemplo, dependiendo de la opción seleccionada en un conjunto de RadioButtons, determinadas entradas de texto pueden ser innecesarias, o tener valores predefinidos o predecibles, que no necesitan ser editados. Otro ejemplo puede ser un programa en el que, dependiendo del nivel de privilegios de un usuario, determinados valores puedan o no ser modificados.



Además, si queremos ser precisos, un control edit de sólo lectura no es en todo equivalente a un control estático. Por ejemplo, el texto del control edit siempre puede ser marcado y copiado al portapapeles, algo que no se puede hacer con los textos de los controles estáticos.

Para modificar esta opción para un control edit se envía un mensaje [EM\\_SETREADONLY](#).

Para averiguar si un control edit tiene el estilo [ES\\_READONLY](#) se debe usar la función [GetWindowLong](#) usando la constante [GWL\\_STYLE](#).

Para ilustrar esto, modificaremos el ejemplo 5 para añadir un checkbox que active y desactive el control edit. Empezaremos por modificar la definición del diálogo en el fichero de recursos:

```
DialogoPrueba DIALOG 0, 0, 118, 58
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
BEGIN
    CONTROL "Sólo lectura", ID_ACTIVAR, "BUTTON",
        BS_AUTOCHECKBOX | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        12, 4, 60, 12
    CONTROL "Texto:", -1, "STATIC",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        8, 20, 28, 8
    CONTROL "", ID_TEXTO, "EDIT",
        ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_TABSTOP,
        36, 20, 76, 12
    CONTROL "Aceptar", IDOK, "BUTTON",
        BS_DEFPUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        8, 36, 45, 14
    CONTROL "Cancelar", IDCANCEL, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        61, 36, 45, 14
END
```

Añadiremos un identificador para el control CheckBox en win050.h:

```
#define ID_ACTIVAR 101
```

Añadiremos un dato a la estructura de datos del cuadro de diálogo para almacenar el estado del CheckBox:

```
typedef struct stDatos {  
    char Texto[80];  
    BOOL Estado;  
} DATOS;
```

Por supuesto, asignaremos un valor inicial a ese estado en el procedimiento de ventana:

```
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT msg, WPARAM  
wParam, LPARAM lParam)  
{  
    static HINSTANCE hInstance;  
    /* Variables para diálogo */  
    static DATOS Datos;  
  
    switch (msg)                                /* manipulador del mensaje  
*/  
    {  
        case WM_CREATE:  
            hInstance = ((LPCREATESTRUCT)lParam)->hInstance;  
            /* Inicialización de los datos de la aplicación  
*/  
            strcpy(Datos.Texto, "Inicial");  
            Datos.Estado = FALSE;  
            return 0;  
        ...  
    }
```

Y modificaremos el procedimiento de diálogo para tratar este nuevo control:

```

BOOL CALLBACK DlgProc(HWND hDlg, UINT msg, WPARAM wParam,
LPARAM lParam)
{
    static DATOS *Datos;

    switch (msg)                /* manipulador del mensaje
*/
    {
        case WM_INITDIALOG:
            SendDlgItemMessage(hDlg, ID_TEXTO, EM_LIMITTEXT,
80, 0L);
            Datos = (DATOS *)lParam;
            SetDlgItemText(hDlg, ID_TEXTO, Datos->Texto);
            /* Acticar el estilo según el valor actual de
estado */
            SendMessage(GetDlgItem(hDlg, ID_TEXTO),
EM_SETREADONLY, Datos->Estado, 0);
            /* Aplicar el estado actual del CheckBox */
            CheckDlgButton(hDlg, ID_ACTIVAR,
                Datos->Estado ? BST_CHECKED : BST_UNCHECKED);
            SetFocus(GetDlgItem(hDlg, ID_TEXTO));
            return FALSE;
        case WM_COMMAND:
            switch(LOWORD(wParam)) {
                case ID_ACTIVAR:
                    Datos->Estado = !Datos->Estado;
                    SendMessage(GetDlgItem(hDlg, ID_TEXTO),
EM_SETREADONLY, Datos->Estado, 0);
                    break;
                case IDOK:
                    GetDlgItemText(hDlg, ID_TEXTO, Datos-
>Texto, 80);
                    EndDialog(hDlg, FALSE);
                    break;
                case IDCANCEL:
                    EndDialog(hDlg, FALSE);
                    break;
            }
            return TRUE;
    }
    return FALSE;
}

```

## Ejemplo 50

## Leer contraseñas

A veces no nos interesa que el texto que se introduce en un control edit no se muestre en pantalla de forma que pueda ser reconocido. El caso más frecuente es cuando se introducen contraseñas. En esos casos se hace que el texto introducido se sustituya por otros caracteres. El usuario que introduce la contraseña sabe qué escribe, porque es el que maneja el teclado, pero una persona que observe este proceso no podrá reconocer el texto en pantalla, y le resultará complicado deducir el texto mirando el teclado.

Para que un control edit se comporte de este modo bastará con activar el estilo `ES_PASSWORD` al crear el control.

Las funciones para asignar valores iniciales o recuperarlos del control funcionarán igual que con los controles normales, el estilo sólo afecta al modo en que se visualiza el texto, no a su contenido.

Por defecto, el carácter que se usa para sustituir los introducidos es el asterisco, pero esto se puede modificar usando el mensaje `EM_SETPASSWORDCHAR`. Si se utiliza un carácter nulo se mostrará el texto que introduzca el usuario.

También podemos usar el mensaje `EM_GETPASSWORDCHAR` para averiguar el carácter que se usa actualmente para sustituir lo introducidos por el usuario.

Estos mensajes sólo están disponibles para controles edit de una línea.

Crearemos otro programa de ejemplo basado en el ejemplo 5. En este caso añadiremos tres RadioButtons con tres opciones distintas de caracteres: el '\*', el '.' y el nulo.

El primer paso es modificar el fichero de recursos para añadir los tres botones:

```
DialogoPrueba DIALOG 0, 0, 118, 98
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Diálogo de prueba"
```

```

FONT 8, "Helv"
BEGIN
    CONTROL "Grupo 1", ID_GRUPO1, "BUTTON",
        BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP,
        4, 5, 76, 52
    CONTROL "Asteriscos", ID_RADIOBUTTON1, "BUTTON",
        BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        11, 15, 60, 12
    CONTROL "Puntos", ID_RADIOBUTTON2, "BUTTON",
        BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        11, 28, 60, 12
    CONTROL "Visible", ID_RADIOBUTTON3, "BUTTON",
        BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        11, 41, 60, 12
    CONTROL "Texto:", -1, "STATIC",
        SS_LEFT | WS_CHILD | WS_VISIBLE,
        8, 64, 28, 8
    CONTROL "", ID_TEXTO, "EDIT",
        ES_LEFT | ES_PASSWORD | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP,
        36, 64, 76, 12
    CONTROL "Aceptar", IDOK, "BUTTON",
        BS_DEFPUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        8, 80, 45, 14
    CONTROL "Cancelar", IDCANCEL, "BUTTON",
        BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP,
        61, 80, 45, 14
END

```

También añadiremos las constantes necesarias en el fichero win051.h:

```

#define ID_GRUPO1 101
#define ID_RADIOBUTTON1 102
#define ID_RADIOBUTTON2 103
#define ID_RADIOBUTTON3 104

```

Tendremos que modificar la estructura de datos para pasar al procedimiento de diálogo:

```
typedef struct stDatos {
    char Texto[80];
    int Estado;
} DATOS;
```

En el procedimiento de ventana principal iniciaremos los datos miembro:

```
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT msg, WPARAM
wParam, LPARAM lParam)
{
    static HINSTANCE hInstance;
    /* Variables para diálogo */
    static DATOS Datos;

    switch (msg)                /* manipulador del mensaje
*/
    {
        case WM_CREATE:
            hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
            /* Inicialización de los datos de la aplicación
*/
            strcpy(Datos.Texto, "Inicial");
            Datos.Estado = 0;
            return 0;
            break;
        ...
    }
```

Por último, procesaremos los mensajes procedentes de los RadioButtons e iniciaremos los controles en el procedimiento de diálogo:

```
BOOL CALLBACK DlgProc(HWND hDlg, UINT msg, WPARAM wParam,
LPARAM lParam)
{
    static DATOS *Datos;
    char opcion[3] = "*.\000";

    switch (msg)                /* manipulador del mensaje
```

```

*/
{
    case WM_INITDIALOG:
        SendDlgItemMessage(hDlg, ID_TEXTO, EM_LIMITTEXT,
80, 0L);
        Datos = (DATOS *)lParam;
        SetDlgItemText(hDlg, ID_TEXTO, Datos->Texto);
        /* Aplicar el carácter según el valor de estado
*/
        SendMessage(GetDlgItem(hDlg, ID_TEXTO),
EM_SETPASSWORDCHAR, opcion[Datos->Estado], 0);
        /* Activar el radiobutton */
        CheckRadioButton(hDlg, ID_RADIOBUTTON1,
ID_RADIOBUTTON3,
ID_RADIOBUTTON1+Datos->Estado);
        SetFocus(GetDlgItem(hDlg, ID_TEXTO));
        return FALSE;
    case WM_COMMAND:
        switch(LOWORD(wParam)) {
            case ID_RADIOBUTTON1:
            case ID_RADIOBUTTON2:
            case ID_RADIOBUTTON3:
                Datos->Estado = LOWORD(wParam) -
>ID_RADIOBUTTON1;
                SendMessage(GetDlgItem(hDlg, ID_TEXTO),
EM_SETPASSWORDCHAR, opcion[Datos->Estado], 0);
                SetFocus(GetDlgItem(hDlg, ID_TEXTO));
                break;
        }
    ...
}

```

A pesar de que la documentación del API afirma que el control edit se actualiza tan pronto recibe un mensaje [EM\\_SETPASSWORDCHAR](#), lo cierto es que no parece que sea así, de modo que en este ejemplo hemos optado por asignar el foco al control edit, esto obliga a que se actualice su aspecto.

## Ejemplo 51

### Mayúsculas y minúsculas

Disponemos de dos estilos para los controles edit que nos sirven para limitar el tipo de caracteres que se pueden usar. El estilo [ES\\_LOWERCASE](#) convierte cualquier carácter en mayúscula a minúscula. El contenido del control edit será sólo de letras en minúsculas, números y caracteres especiales. El estilo [ES\\_UPPERCASE](#) convierte cualquier carácter en minúscula a mayúsculas.

El estilo no afecta al contenido de los caracteres del control, podemos asignar valores con mayúsculas o minúsculas con cualquiera de los estilos. Estos estilos afectan sólo a los nuevos caracteres introducidos por el usuario.

Los estilos se pueden asignar en el momento de la creación del control, como hemos hecho hasta ahora mediante ficheros de recursos o mediante las funciones [CreateWindow](#) y [CreateWindowEx](#), o se pueden modificar durante la ejecución, usando la función [SetWindowLong](#), y la función [GetWindowLong](#) para obtener el estilo actual:

```
    ActivarEstilo(GetDlgItem(hDlg, ID_TEXTO), estado);
...
void ActivarEstilo(HWND hctrl, int estado)
{
    LONG estiloActual;

    estiloActual = GetWindowLong(hctrl, GWL_STYLE);
    switch(estado) {
        case 0: /* Normal */
            estiloActual &= ~ES_UPPERCASE;
            estiloActual &= ~ES_LOWERCASE;
            break;
        case 1: /* Mayúsculas */
            estiloActual &= ~ES_LOWERCASE;
            estiloActual |= ES_UPPERCASE;
            break;
        case 2: /* Minúsculas */
            estiloActual &= ~ES_UPPERCASE;
            estiloActual |= ES_LOWERCASE;
            break;
    }
}
```



```
SetWindowLong(hctrl, GWL_STYLE, estiloActual);  
}
```

## Ejemplo 52

### Mensajes de notificación

Windows envía un tipo especial de mensajes, denominados *mensajes de notificación*, a la ventana padre de un control edit. Estos mensajes sirven para informar a la aplicación de determinadas circunstancias relativas a un control.

Los mensajes de notificación se reciben a través de un mensaje [WM\\_COMMAND](#). En la palabra de menor peso del parámetro *wParam* se envía el identificador del control. El manipulador del control se envía en el parámetro *lParam* y el código del mensaje de notificación en la palabra de mayor peso de *wParam*.

**Nota:**

En el API de Windows 3.x el código del mensaje de notificación se envía en el parámetro *lParam*. Hay que tener esto en cuenta si se intenta portar código entre estas plataformas.

Veamos a continuación los mensajes de notificación que existen para los controles edit:

### Modificación

Cada vez que el usuario modifica el texto de un control edit, primero se actualiza el contenido del control en pantalla, y a continuación se genera un mensaje [EN\\_CHANGE](#).

### Actualización

Cada vez que el usuario modifica el texto de un control Edit, y antes de que este nuevo texto se muestre en pantalla, Windows envía un mensaje [EN\\_UPDATE](#).

Este mensaje está pensado para permitir a la aplicación redimensionar el tamaño del control en función de su contenido.

## **Falta espacio**

Cuando el control edit no puede conseguir espacio de memoria suficiente para realizar una operación se envía un mensaje de notificación [EN\\_ERRSPACE](#).

## **Desplazamiento horizontal y vertical**

Cuando el usuario hace clic sobre una barra de desplazamiento de un control edit, horizontal o vertical, se envía el mensaje [EN\\_HSCROLL](#) o [EN\\_VSCROLL](#), respectivamente, antes de que la pantalla se actualice.

## **Pérdida y recuperación de foco**

Cada vez que el usuario selecciona otro control se envía un mensaje de notificación [EN\\_KILLFOCUS](#).

Cuando el usuario selecciona un control edit, se envía un mensaje de notificación [EN\\_SETFOCUS](#).

## **Texto máximo**

El mensaje de notificación [EN\\_MAXTEXT](#) se envía si el usuario intenta escribir más caracteres de los especificados para un control edit.

También se envía este mensaje si la anchura de la cadena introducida en el control es mayor de la anchura del control y no se ha especificado el estilo [ES\\_AUTOHSCROLL](#), o si el número total

de líneas a insertar en un control edit multilinea excede la altura del control y no se ha especificado el estilo [ES\\_AUTOVSCROLL](#).

Por ejemplo, para gestionar los mensajes de notificación de un control edit con el identificador ID\_TEXTO, usaríamos un código parecido a este:

```
case WM_COMMAND:
    switch(LOWORD(wParam)) {
        case ID_TEXTO:
            /* Mensajes de notificación */
            switch(HIWORD(wParam)) {
                case EN_MAXTEXT:
                    MessageBox(hwnd, "Imposible insertar
más caracteres", "Control edit", MB_OK);
                    break;
                case EN_ERRSPACE:
                    ...
                case EN_HSCROLL:
                    ...
                case EN_VSCROLL:
                    ...
                case EN_KILLFOCUS:
                    ...
                case EN_SETFOCUS:
                    ...
                case EN_UPDATE:
                    ...
                case EN_CHANGE:
                    ...
            }
        ...
    }
```

## El buffer de texto

Hasta ahora no nos hemos preocupado nunca del espacio de memoria necesario para almacenar y editar el contenido de un control edit. Windows se encarga de crear un buffer, y de aumentar su tamaño si es necesario, hasta cierto límite, dependiendo del tipo de control edit.

En el [capítulo 7](#) vimos que podíamos fijar el límite máximo que el usuario podía editar mediante el mensaje [EM\\_LIMITTEXT](#).

Sin embargo este mensaje no limita el tamaño del buffer. El mensaje no tiene efecto si el control ya contiene más caracteres que el límite establecido, y sigue siendo posible insertar más caracteres usando el mensaje [WM\\_SETTEXT](#). De hecho, este mensaje no debería usarse, ya que ha sido sustituido por [EM\\_SETLIMITTEXT](#).

Para limitar el tamaño del buffer se usa el mensaje [EM\\_SETLIMITTEXT](#), y para obtener el valor del tamaño del buffer se usa el mensaje [EM\\_GETLIMITTEXT](#).

En versiones de Windows de 16 bits es posible asumir, por parte de nuestra aplicación, todas las operaciones de control del buffer de memoria asociado a un control edit multilinea. Para ello, lo primero que debemos hacer es crear el control edit en una ventana que use el estilo [DS\\_LOCALEEDIT](#). Además disponemos de los mensajes [EM\\_GETHANDLE](#) y [EM\\_SETHANDLE](#), para obtener un manipulador de memoria local del buffer del control, o asignar uno nuevo, respectivamente.

El proceso consiste en:

- Obtener un manipulador del buffer local actual, mediante [EM\\_GETHANDLE](#).
- Liberar ese buffer usando la función [LocalFree](#).
- Crear un nuevo buffer local, usando [LocalAlloc](#).
- Asignar el nuevo manipulador de memoria al control, mediante el mensaje [EM\\_SETHANDLE](#).

Cada vez que el buffer se quede pequeño recibiremos un mensaje de notificación [EN\\_ERRSPACE](#).

Este proceso es inútil en el API de 32 bits, ya que en este caso toda la memoria pertenece al espacio de direcciones de memoria virtual, y no hay distinción entre memoria local y global.

## Controles multilinea

Hasta ahora sólo hemos trabajado con controles edit de una línea, pero también es posible crear controles edit multilínea. Para ello bastará con crearlos con el estilo `ES_MULTILINE`. Pero estos controles tienen algunas peculiaridades que los hace algo más complicados de usar que los de una línea.

Para empezar, cuando se ejecuta un cuadro de diálogo, la tecla ENTER tiene el efecto de activar el botón por defecto. Esto nos crea un problema con los controles edit multilínea, ya que no podremos usar la tecla de [ENTER] para insertar un retorno de línea. Para evitar este comportamiento por defecto en los cuadros de diálogo se usa el estilo `ES_WANTRETURN`. Este estilo hace que las pulsaciones de la tecla ENTER, cuando el control tiene el foco del teclado, se conviertan en retornos de línea, y no se active el botón por defecto.

Otro detalle importante es que con frecuencia el texto no va a caber en el área visible del control, por lo que tendremos que desplazar el contenido tanto horizontal como verticalmente.

Para lograr esto disponemos, por una parte, de dos estilos propios de los controles edit: `ES_AUTOHSCROLL` y `ES_AUTOVSCROLL`. Cuando se activan estos estilos el texto se desplaza de forma automática en sentido horizontal o vertical, respectivamente, cada vez que el usuario llegue a un borde del área del control mientras escribe texto.

Además de esta posibilidad tenemos una segunda que consiste en añadir las barras de desplazamiento. Estas barras se añaden con los estilos de ventana `WS_HSCROLL` y `WS_VSCROLL`, respectivamente, y activan de forma automática los dos estilos anteriores: `ES_AUTOHSCROLL` y `ES_AUTOVSCROLL`.

La diferencia es que de esta segunda forma se muestran las barras, y de la primera no.

## **Iniciar controles multilínea**

Otra dificultad añadida a la hora de usar estos controles es la inicialización. Las líneas dentro de un control edit multilinea se separan con dos retornos de línea y un avance de línea, es decir, dos caracteres '\r' y uno '\n', a esta secuencia se le denomina una *ruptura de línea blanda*. Por otra parte, si queremos convertir un retorno de línea normal en una ruptura de línea blanda, hay que saber que Windows añade de forma automática un carácter '\r' cada vez que se añade un carácter '\n', es decir, Windows sustituye el carácter '\n' por la secuencia "\r\n".

En cualquier caso, esto nos obliga a hacer un tratamiento de cada línea del texto de inicialización para sustituir las secuencias "\n" o "\r\n" por otra "\r\n".

Tenemos, pues, tres opciones a la hora de inicializar controles edit multilinea.

Una consiste en crear un buffer de texto con el contenido, sustituyendo los cambios de línea por rupturas blandas, y asignar el texto al control mediante un mensaje [WM\\_SETTEXT](#).

```
void AsignarTexto(HWND hctrl, char *texto) {
    char* buffer;

    /* Crear buffer */
    buffer = (char *)malloc(strlen(texto)+1);
    buffer[0] = 0;

    /* Hacer la lectura */
    SustituirCambiosDeLinea(texto, buffer);

    SendMessage(hctrl, WM_SETTEXT, 0, (LPARAM)buffer);
    free(buffer);
}
```

Otra es usar el mismo buffer, creado con la función [LocalAlloc](#), y asignar ese buffer al control edit directamente.

```
void AsignarTexto(HWND hctrl, char *texto) {
```

```

char* buffer;
HLOCAL hloc;

/* Obtener manipulador de buffer actual: */
hloc = (HLOCAL)SendMessage(hctrl, EM_GETHANDLE, 0, 0);
/* Liberar buffer actual */
LocalFree(hloc);
/* Crear un buffer nuevo */
hloc = LocalAlloc(LMEM_MOVEABLE, 1+1);

/* Bloquear el buffer para su uso */
buffer = (char *)LocalLock(hloc);
buffer[0] = 0;

/* Hacer la lectura */
SustituirCambiosDeLinea(texto, buffer);

/* Desbloquear buffer */
LocalUnlock(hloc);

/* Asignar el nuevo buffer al control edit */
SendMessage(hctrl, EM_SETHANDLE, (WPARAM)hloc, 0);
}

```

Una tercera opción consiste en enviar el contenido del control carácter a carácter, mediante mensajes [WM\\_CHAR](#). La ventaja de este método es que no hay que insertar rupturas blandas, ya que el sistema lo hace por nosotros:

```

void AsignarTexto(HWND hctrl, char *texto) {
    int i;

    for(i = 0; i < strlen(texto); i++)
        SendMessage(hctrl, WM_CHAR, (WPARAM)texto[i], 0);
}

```

## Mensajes para controles multilínea

Disponemos de varios mensajes útiles cuando se trabaja con controles edit multilínea, veremos ahora algunos de ellos:

Para obtener el número de caracteres en un control edit se usa el mensaje [WM\\_GETTEXTLENGTH](#). Los caracteres están indexados empezando en cero, pero es muy importante tener en cuenta que en esta indexación no se incluyen los caracteres de los retornos de línea.

Otro mensaje, más específico de controles edit, y que también se usa para calcular longitudes es [EM\\_LINELENGTH](#). Este mensaje está orientado a obtener longitudes de líneas en controles multilínea.

El mensaje [EM\\_LINELENGTH](#) requiere un parámetro, que es el índice de un carácter, y devolverá la longitud de la línea a la que pertenece ese carácter. Si ese índice es -1, se devolverá la longitud del texto de las líneas con texto seleccionado, excluyendo la longitud del propio texto seleccionado.

```
int longitud;

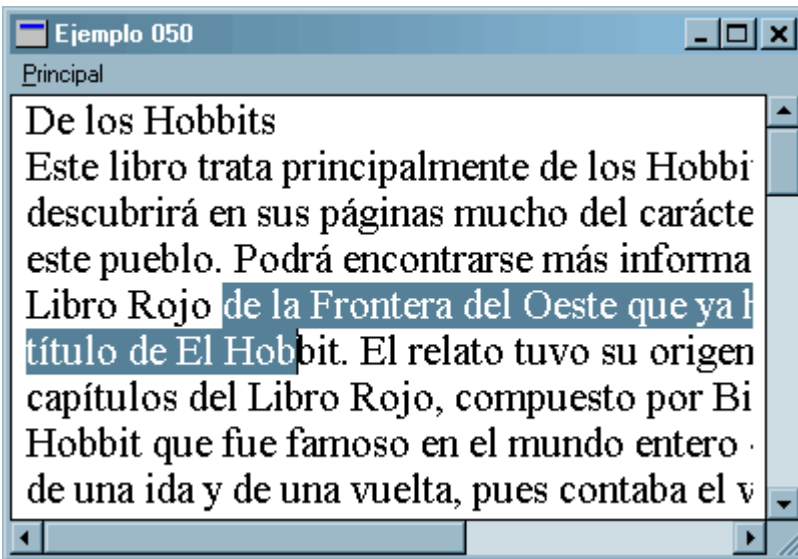
/* Longitud de la o las líneas con texto seleccionado,
excluyendo el texto seleccionado */
longitud = SendMessage(hctrl, EM_LINELENGTH, (WPARAM)-1,
0);
sprintf(mensaje, "Longitud = %d", longitud);
MessageBox(hwnd, mensaje, "Control edit multilínea",
MB_OK);
```

En el ejemplo anterior, si recuperamos la longitud del texto con el índice -1, se calculará a partir de los caracteres no seleccionados en la quinta y sexta línea, ya que la selección ocupa ambas líneas.

Los mensajes [WM\\_GETTEXTLENGTH](#) y [EM\\_LINELENGTH](#) funcionan tanto con controles edit de una línea como con los multilínea.

El mensaje [EM\\_GETFIRSTVISIBLELINE](#) obtiene el índice de la primera línea visible en un control multilínea, empezando en cero, o el índice del primer carácter visible en un control edit de una línea.





### Selección múltiple

```
int pos;

pos =
SendMessage(hctrl
,
EM_GETFIRSTVISIBL
ELINE, 0, 0);

sprintf(mensaje,
"Primera línea
visible = %d",
pos);

MessageBox(hwnd,
mensaje, "Control
edit multilínea",
MB_OK);
```

El mensaje `EM_GETLINE` sirve para obtener el contenido de una línea determinada. Para usar este mensaje se pasa en wParam el índice de la línea a leer, y en lParam un puntero al buffer que recibirá el contenido de la línea. Cuando se envía el mensaje hay que colocar en la primera palabra de ese buffer el tamaño máximo de la cadena a leer:

```
char linea[512];
int longitud;

*(WORD*)linea = 512; /* Longitud del buffer en primera
palabra */
longitud = SendMessage(hctrl, EM_GETLINE, (WPARAM)1,
(LPARAM)linea);
MessageBox(hwnd, linea, "Línea 1", MB_OK);
```

El mensaje `EM_GETLINECOUNT` nos devuelve el número de líneas total que contiene un control edit multilínea:

```

char mensaje[512];
int nLineas;

nLineas = SendMessage(hctrl, EM_GETLINECOUNT, 0, 0);
sprintf(mensaje, "Número de líneas = %d", nLineas);
MessageBox(hwnd, mensaje, "Contol edit multilínea",
MB_OK);

```

Los mensajes [EM\\_GETLINE](#) y [EM\\_GETLINECOUNT](#), usados de forma conjunta, nos permiten leer y tratar (por ejemplo guardar en un fichero), el contenido de un control edit multilínea, sin preocuparnos de las rupturas de línea blandas:

```

void Guardar(HWND hctrl, char *fichero) {
    FILE *fs;
    int nLineas, longitud, i;
    char linea[1024];

    fs = fopen(fichero, "w");
    if(fs) {
        nLineas = SendMessage(hctrl, EM_GETLINECOUNT, 0, 0);
        for(i = 0; i < nLineas; i++) {
            *(WORD*)linea = 1024;
            longitud = SendMessage(hctrl, EM_GETLINE,
(WPARAM)i, (LPARAM)linea);
            linea[longitud] = 0;
            fprintf(fs, "%s\n", linea);
        }
        fclose(fs);
    }
}

```

El mensaje [EM\\_LINEINDEX](#) se usa para averiguar el índice del primer carácter de una línea determinada:

```

int pos;

pos = SendMessage(hctrl, EM_LINEINDEX, (WPARAM)1, 0);

```

```
sprintf(mensaje, "Índice del primer carácter de la línea  
1 = %d", pos);  
MessageBox(hwnd, mensaje, "Control edit multilínea",  
MB_OK);
```

El mensaje [EM\\_LINEFROMCHAR](#) devuelve el índice de la línea que contiene el carácter determinado por un índice dado:

```
int pos;  
  
pos = SendMessage(hctrl, EM_LINEFROMCHAR, (WPARAM)95, 0);  
sprintf(mensaje, "Índice de la línea que contiene el  
carácter 95 = %d", pos);  
MessageBox(hwnd, mensaje, "Control edit multilínea",  
MB_OK);
```

Los mensajes [EM\\_LINEINDEX](#) y [EM\\_LINEFROMCHAR](#) sólo son válidos para controles edit multilínea.

## Ejemplo 53

### Operaciones sobre selecciones de texto

El usuario puede seleccionar una parte del texto incluido en un control edit, bien usando el ratón (que será lo más frecuente), o bien mediante el teclado, (manteniendo pulsada la tecla de mayúsculas y desplazando el cursor mediante el las teclas de movimiento del cursor).

Estamos acostumbrados ya a las operaciones frecuentes que se pueden hacer sobre una selección: copiar, cortar, borrar o pegar. Veamos ahora cómo podemos realizar estas operaciones en nuestros controles edit.

Cualquier control edit procesará los mensajes [WM\\_CUT](#), [WM\\_COPY](#), [WM\\_CLEAR](#) y [WM\\_PASTE](#) que reciba.

En el caso del mensaje `WM_CUT`, se copiará el texto seleccionado en el portapapeles y después se eliminará.

En el caso del mensaje `WM_COPY`, el texto seleccionado se copiará en el portapapeles.

En el caso del mensaje `WM_CLEAR`, el texto seleccionado se eliminará, sin copiarse en el portapapeles.

Y en el caso del mensaje `WM_PASTE`, el texto que esté en el portapapeles se copiará en la posición actual del caret en el control edit o sustituyendo al texto seleccionado, si existe.

Estas acciones están ya implementadas en el procedimiento de ventana de la clase "EDIT", por lo que bastará con enviar cualquiera de estos mensajes a un control para que funcionen.

Es más, las versiones actuales de Windows desplegarán un menú contextual al pulsar el botón derecho del ratón sobre cualquier control edit. Ese menú contendrá estos cuatro comandos: cortar, copiar, pegar y eliminar, y otros dos: deshacer y seleccionar todo.

También funcionan, cuando el control edit tiene el foco, las combinaciones de teclas para estas seis acciones: [control]+x para cortar, [control]+c para copiar, [control]+v para pegar y [supr] para borrar y [control]+z para deshacer.

Por nuestra parte, podemos añadir estos mensajes a nuestros menús, y enviarlos al control edit que tenga el foco en el momento en que se seleccionen.

Además, disponemos de otros mensajes para controlar la selección de texto.

El mensaje `EM_GETSEL` nos sirve para obtener los índices de los caracteres correspondientes al inicio y final de la selección actual. Más concretamente, obtendremos el índice del primer carácter de la selección y el del primero no seleccionado a continuación de la selección. Por ejemplo, si la selección incluye los caracteres 10º al 15º de texto de un control, obtendremos los valores 9 y 15:

Texto de `ejemplo` para ilustrar la selección.  
00000000001111111111222222222233333333334444

01234567890123456789012345678901234567890123

```
int inicio, final;

SendMessage(hctrl, EM_GETSEL, (WPARAM)&inicio,
(LPARAM)&final);
sprintf(mensaje, "Selección actual de %d a %d", inicio,
final);
MessageBox(hwnd, mensaje, "Ejemplo de control edit",
MB_OK);
```

De forma simétrica, podemos seleccionar una parte del texto mediante el mensaje [EM\\_SETSEL](#). Si seleccionamos una parte del texto que no es visible en pantalla, Windows no lo mostrará de forma automática, y el caret quedará fuera de la parte visible del control. Si queremos que el caret sea visible debemos usar el mensaje [EM\\_SCROLLCARET](#):

```
SendMessage(hctrl, EM_SETSEL, 1500, 1554);
SendMessage(hctrl, EM_SCROLLCARET, 0, 0);
```

Por último, el mensaje [EM\\_REPLACESEL](#) nos permite sustituir el texto actualmente seleccionado por otro:

```
char nuevotexto[64];

strcpy(nuevotexto, "TEXTO SUSTITUIDO");
SendMessage(hctrl, EM_REPLACESEL, TRUE,
(LPARAM)nuevotexto);
```

El parámetro wParam de este mensaje es un valor booleano que indica si la operación de sustitución puede ser deshecha o no. **TRUE** significa que se podrá deshacer, **FALSE** que no.

## Selecciones siempre visibles

Por defecto, si no indicamos lo contrario, el texto seleccionado en un control edit, permanecerá resaltado sólo mientras el control tiene el foco del teclado. Sin embargo, si creamos el control con el estilo [ES\\_NOHIDSEL](#), el resaltado permanece aunque el control pierda el foco.

## Deshacer cambios (undo)

Otra tarea que realiza Windows con los controles edit es la de deshacer cambios, aunque el soporte para esta tarea es limitado, ya que sólo es posible deshacer el último cambio realizado en un control edit.

Al igual que sucede con los mensajes [WM\\_CUT](#), [WM\\_COPY](#), [WM\\_PASTE](#) y [WM\\_CLEAR](#), el mensaje [EM\\_UNDO](#) se procesa de forma automática por el procedimiento de ventana del control edit. Por lo tanto bastará con enviar este mensaje a un control edit, y si es posible, se deshará el último cambio realizado.

```
SendMessage(hwnd, EM_UNDO, 0, 0);
```

Disponemos de otros dos mensajes relacionados con estas operaciones. El mensaje [EM\\_CANUNDO](#) nos permite averiguar si es posible deshacer alguna operación de edición. Es decir, si al enviar el mensaje [EM\\_UNDO](#) se deshará algún cambio.

Por último, el mensaje [EM\\_EMPTYUNDOBUFFER](#) vacía el buffer de deshacer, es decir, anula la posibilidad de deshacer cualquier operación de edición.

Para los controles edit sólo se almacena una operación de deshacer, es decir, que enviar repetidamente el mensaje [EM\\_UNDO](#) sólo deshará la última operación de edición, no las anteriores. Si lo último que se hizo fue eliminar un fragmento de texto, el primer mensaje [EM\\_UNDO](#) restaurará el texto eliminado, el segundo

volverá a eliminarlo, el tercero lo restaurará de nuevo, y así sucesivamente.

Para operaciones de deshacer más elaboradas deberemos programar nosotros mismos las rutinas necesarias.

## Modificación del texto

Windows mantiene una bandera para cada control edit que indica si su contenido ha sido modificado por el usuario o no.

Esta bandera nos permite tomar decisiones en función de si el contenido de un control edit ha sido o no modificado. Por ejemplo, si el contenido de un control correspondiente a un editor de texto no se ha modificado, no tendrá sentido leerlo y actualizar el fichero original.

Windows desactiva esta bandera automáticamente al crear el control, y la activa cada vez que el usuario edita el contenido del control.

Para leer el valor actual de esta bandera se usa el mensaje [EM\\_GETMODIFY](#):

```
BOOL modif;

modif = SendMessage(hctrl, EM_GETMODIFY, 0, 0);
if(modif) strcpy(mensaje, "Texto modificado");
else strcpy(mensaje, "Texto no modificado");
MessageBox(hwnd, mensaje, "Ejemplo de control edit",
MB_OK);
```

En ocasiones nos interesará volver a desactivar esta bandera, por ejemplo, cuando hemos guardado el contenido actual de nuestro editor en disco, consideraremos que el contenido actual no ha sido modificado, puesto que es el mismo que en el fichero. Para modificar el valor de la bandera se usa el mensaje [EM\\_SETMODIFY](#).

También podemos combinar el estado de esta bandera con el mensaje de notificación **EN\_CHANGE**. Esto es lo que se suele hacer para mostrar el estado de modificación de un texto en un editor. Cada vez que se recibe un mensaje de notificación **EN\_CHANGE**, se consulta la bandera de modificación, y en función de su valor, se pone la marca que indica si es necesario guardar el contenido o no. También se pueden inhibir las opciones de guardar si la bandera de modificación no está activa.

```
case WM_COMMAND:
    switch(LOWORD(wParam)) {
        case CM_GUARDAR:
            Guardar(hCtrl, "texto.txt");
            SendMessage(hCtrl, EM_SETMODIFY, FALSE, 0);
            ActualizarMenu(hwnd, hCtrl);
            break;
        case ID_TEXTO:
            if(HIWORD(wParam) == EN_CHANGE)
                if(EnableMenuItem(hCtrl, EM_GETMODIFY, 0,
0))
                    EnableMenuItem(GetMenu(hwnd),
CM_GUARDAR, MF_BYCOMMAND | MF_ENABLED);
            break;
        ...
    }

void ActualizarMenu(HWND hwnd, HWND hCtrl) {
    if(SendMessage(hCtrl, EM_GETMODIFY, 0, 0))
        EnableMenuItem(GetMenu(hwnd), CM_GUARDAR, MF_BYCOMMAND
| MF_ENABLED);
    else
        EnableMenuItem(GetMenu(hwnd), CM_GUARDAR, MF_BYCOMMAND
| MF_GRAYED);
}
```

En este fragmento vemos cómo manipular el mensaje de notificación **EN\_CHANGE** para verificar la bandera de modificación, y activar la opción de menú de "guardar" cuando el contenido del control haya sido modificado.

El procesamiento del mensaje de "guardar" guarda el contenido del control edit, y a continuación elimina la bandera de modificación



y actualiza el estado del menú.

## Márgenes y tabuladores

Si no indicamos nada, el control edit usará toda la superficie de su área de cliente para mostrar el texto. Pero podemos cambiar esto de varias formas, definiendo los márgenes izquierdo y derecho, o especificando un rectángulo, dentro del área de cliente, que se usará para mostrar el texto.

El mensaje [EM\\_SETMARGINS](#) nos permite fijar los márgenes izquierdo y/o derecho del texto dentro del control edit. En el parámetro wParam indicamos qué márgenes vamos a definir, y en qué unidades se expresan. Para ello podemos combinar los valores [EC\\_LEFTMARGIN](#), [EC\\_RIGHTMARGIN](#) y [EC\\_USEFONTINFO](#). El primero para definir el margen izquierdo, el segundo para definir el derecho, y el tercero para indicar que usaremos la anchura del carácter "A" de la fuente actual para el margen izquierdo, y el del carácter "C" para el derecho. Si no usamos este valor, el parámetro lParam indicará la anchura en pixels.

El parámetro lParam indica el margen izquierdo en la palabra de menor peso, y el derecho en la de mayor peso. Para combinar estos valores se puede usar la macro [MAKELONG](#).

```
SendMessage(hwnd, EM_SETMARGINS,  
            EC_LEFTMARGIN | EC_RIGHTMARGIN, MAKELONG(50, 30));
```

El mensaje [EM\\_GETMARGINS](#) se puede usar para recuperar los valores de los márgenes actuales de un control edit.

Otra opción es usar el mensaje [EM\\_SETRECT](#) para definir el rectángulo que se usará para delimitar el texto. En este mensaje se usa el parámetro lParam para indicar un puntero a una estructura [RECT](#) que define el rectángulo delimitador.

El mensaje [EM\\_SETRECTNP](#) es idéntico, con la diferencia de que no se actualiza el control edit para reflejar el nuevo aspecto del control.

```
GetClientRect(hctrl, &re);  
re.left += 60;  
re.top += 20;  
re.right -= 40;  
re.bottom -= 40;  
SendMessage(hctrl, EM_SETRECT, 0, (LPARAM)&re);
```

La ventaja de este método es que nos permite definir los márgenes superior e inferior, además del derecho e izquierdo. La desventaja es que estas definiciones no son permanentes, al contrario que en el caso del mensaje [EM\\_SETMARGINS](#). De modo que si cambiamos el tamaño de la ventana del control, deberemos volver a definir los márgenes.

Para recuperar el rectángulo delimitador actual de un control edit se puede usar el mensaje [EM\\_GETRECT](#).

Por último, hablaremos de los tabuladores. En los controles edit multilínea, los tabuladores no se despliegan como un número fijo de espacios, sino que corresponden a distancias fijas con respecto al borde izquierdo de la ventana del control. Cada vez que se introduce un carácter tabulador, se añade el espacio necesario para desplazar el caret a la siguiente posición del tabulador. Recordemos que los controles edit pueden usar fuentes de espacio proporcional, y que el uso principal de los tabuladores es crear tablas, por lo tanto, este es el comportamiento más lógico.

Podemos fijar las marcas de tabulación mediante el mensaje [EM\\_SETTABSTOPS](#). Estas distancias se miden con respecto al borde izquierdo, y se expresan en unidades de diálogo.

Si sólo indicamos una marca de tabulación, todas las marcas se situarán a distancias iguales, si indicamos más, las primeras n

marcas se situarán a las distancias indicadas, y el resto serán equidistantes.

```
DWORD lista[10] = {10,25,40,65,95,130,160,200,250,360};  
SendMessage(hctrl, EM_SETTABSTOPS, 10, (LPARAM)&lista);
```

## Desplazar texto

Ya vimos más arriba, cuando hablamos del mensaje para seleccionar texto, que podemos desplazar el contenido de un control edit hasta el punto donde se encuentre el caret, con el fin de hacerlo visible. Para esto usamos el mensaje [EM\\_SCROLLCARET](#):

```
...  
SendMessage(hctrl, EM_SETSEL, 1500, 1559);  
SendMessage(hctrl, EM_SCROLLCARET, 0, 0);  
...
```

Pero disponemos de otros dos mensajes para desplazar el contenido de un control edit. Por una parte, el mensaje [EM\\_LINESCROLL](#), nos permite desplazar el texto verticalmente un número de líneas especificado, y desplazar horizontalmente el número de caracteres especificado:

```
/* Desplazar texto 20 caracteres a la derecha  
   y 10 líneas hacia abajo */  
SendMessage(hctrl, EM_LINESCROLL, 20, 10);
```

El otro mensaje es [EM\\_SCROLL](#), que equivale a usar el mensaje [WM\\_VSCROLL](#), y permite desplazar el texto verticalmente, línea a línea o página a página:

```

case CM_PAGINAARRIBA:
    SendMessage(hctrl, EM_SCROLL, SB_PAGEUP, 0);
    break;
case CM_LINEAARRIBA:
    SendMessage(hctrl, EM_SCROLL, SB_LINEUP, 0);
    break;
case CM_LINEAABAJO:
    SendMessage(hctrl, EM_SCROLL, SB_LINEDOWN, 0);
    break;
case CM_PAGINAABAJO:
    SendMessage(hctrl, EM_SCROLL, SB_PAGEDOWN, 0);
    break;

```

## Ejemplo 54

### Caracteres y posiciones

Por último, disponemos de dos mensajes que relacionan los puntos físicos de la pantalla con los caracteres que ocupan esas posiciones. El mensaje [EM\\_CHARFROMPOS](#) nos devuelve el índice del carácter situado en las coordenadas especificadas.

Las coordenadas se proporcionan en el parámetro lParam, la x en la palabra de menor peso, y la y en la de mayor peso. Para crear un valor [LPARAM](#) a partir de las coordenadas podemos usar la macro [MAKELPARAM](#).

El valor obtenido contiene en la palabra de mayor peso el índice de la línea, y en la de menor peso, el índice del carácter:

```

int indice, indicecaracter;
...
indice = SendMessage(hctrl, EM_CHARFROMPOS, 0,
MAKELPARAM(86,52));
indicecaracter = LOWORD(indice);
/* Seleccionar carácter en esa coordenada */
SendMessage(hctrl, EM_SETSEL, indicecaracter,
indicecaracter+1);

```

El mensaje `EM_POSFROMCHAR` es el inverso al anterior: a partir de un índice de carácter nos devuelve las coordenadas de ese carácter en pantalla.

Para ello indicaremos en el parámetro `wParam` el índice del carácter, el valor de retorno contiene las coordenadas correspondientes, la `x` en la palabra de menor peso y la `y` en la de mayor.

```
LRESULT punto;  
char mensaje[128];  
...  
punto = SendMessage(hwnd, EM_POSFROMCHAR, 153, 0);  
sprintf(mensaje, "Coordenadas del carácter 153: "  
        "(%d, %d)", LOWORD(punto), HIWORD(punto));  
MessageBox(hwnd, mensaje, "EM_POSFROMCHAR", MB_OK);
```

## Ejemplo 55

# Capítulo 40 Control List box avanzado

## Insertar controles list box durante la ejecución

Al igual que vimos con los controles edit, también es posible insertar controles list box durante la ejecución. En el caso del control list box tendremos que insertar una ventana de la clase "LISTBOX". Para insertar el control también usaremos las funciones [CreateWindow](#) y [CreateWindowEx](#).

```
    HWND hctrl;
...
    case WM_CREATE:
        hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
        /* Insertar control Edit */
        hctrl = CreateWindowEx(
            0,
            "LISTBOX",          /* Nombre de la clase */
            "",                 /* Texto del título, no tiene
*/
            LBS_STANDARD | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP, /* Estilo */
            9, 19,             /* Posición */
            104, 99,           /* Tamaño */
            hwnd,              /* Ventana padre */
            (HMENU)ID_LISTA,   /* Identificador del control
*/
            hInstance,         /* Instancia */
            NULL);             /* Sin datos de creación de
ventana */
        /* Inicialización de los datos de la aplicación
*/
        SendMessage(hctrl, LB_ADDSTRING, 0,
```

```
(LPARAM) "Cadena nº 1");  
    SendMessage(hctrl, LB_ADDSTRING, 0,  
(LPARAM) "Cadena nº 4");  
    SendMessage(hctrl, LB_ADDSTRING, 0,  
(LPARAM) "Cadena nº 3");  
    SendMessage(hctrl, LB_ADDSTRING, 0,  
(LPARAM) "Cadena nº 2");  
    SendMessage(hctrl, LB_SELECTSTRING, (UINT)-1,  
(LPARAM) Datos.Item);  
    SetFocus(hctrl);  
    return 0;
```

Como vemos, usamos los mismos valores que en el fichero de recursos: identificador, clase de ventana (en este caso "LISTBOX"), estilo, posición y dimensiones.

Al igual que en el caso del control edit, el identificador del control se suministra a través del parámetro *hMenu*, por lo que será necesario hacer un casting del valor del identificador a [HMENU](#).

Ahora será nuestro procedimiento de ventana, (si el control se ha insertado en una ventana), el encargado de procesar los mensajes procedentes del control. Recordemos que en los ejemplos que hemos visto hasta ahora esto lo hacía el procedimiento de diálogo.

## Cambiar la fuente de un control list box

También es posible modificar la fuente de un control list box enviando un mensaje [WM\\_SETFONT](#). El lugar apropiado es, por supuesto, al procesar el mensaje [WM\\_INITDIALOG](#) cuando se inicie un cuadro de diálogo, o al procesar el mensaje [WM\\_CREATE](#) cuando se inicie una ventana,.

En el parámetro *wParam* pasamos un manipulador de fuente, y usaremos la macro [MAKELPARAM](#) para crear un valor [LPARAM](#), en el que especificaremos la opción de repintar el control, que se almacena en la palabra de menor peso de [LPARAM](#).

Esto nos permite modificar la fuente durante la ejecución, reflejando los cambios en pantalla.

```

static HFONT hfont;
...
        hfont = CreateFont(24, 0, 0, 0, 300,
                           FALSE, FALSE, FALSE, DEFAULT_CHARSET,
                           OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
                           PROOF_QUALITY, DEFAULT_PITCH | FF_ROMAN,
                           "Times New Roman");
        SendMessage(hctrl, WM_SETFONT, (WPARAM)hfont,
MAKELPARAM(TRUE, 0));
...
        case WM_DESTROY:
            DeleteObject(hfont);
...

```

En el caso de crear una fuente especial para nuestros controles, debemos recordar destruirla cuando ya no sea necesaria, generalmente al destruir la ventana.

Por supuesto, también podemos usar una fuente de stock:

```

        hfont = (HFONT)GetStockObject( DEFAULT_GUI_FONT
);
        SendMessage(hctrl, WM_SETFONT, (WPARAM)hfont,
MAKELPARAM(TRUE, 0));

```

## Cambiar los colores de un control list box

Para terminar, también podemos personalizar más nuestros controles list box, cambiando los colores del texto y del fondo. Para ello deberemos procesar el mensaje [WM\\_CTLCOLORLISTBOX](#).

Este mensaje se envía a la ventana padre del control justo antes de que el sistema lo vaya a dibujar, y nos permite cambiar los colores del texto y fondo. Para ello nos suministra en el parámetro wParam un manipulador del contexto de dispositivo del control, y en lParam un manipulador del control, mediante el cual podemos saber a qué control concreto se refiere el mensaje.



El valor de retorno, si se procesa este mensaje, debe ser un manipulador de pincel con el color de fondo del control.

```
static HBRUSH pincel;
HWND hcrtl;

switch (msg)                                /* manipulador del mensaje
*/
{
    case WM_CREATE:
        hcrtl = CreateWindowEx(...);
        pincel = CreateSolidBrush( RGB(0,255,0) );
        SetFocus(hcrtl);
        return 0;
    case WM_CTLCOLORLISTBOX:
        SetBkColor( (HDC)wParam, RGB(0,255,0) );
        SetTextColor( (HDC)wParam, RGB(255,255,255) );
        return (LRESULT)pincel;
    case WM_DESTROY:
        DeleteObject(pincel);
        PostQuitMessage(0);    /* envía un mensaje
WM_QUIT a la cola de mensajes */
        break;
    ...
}
```

## Ejemplo 56

### Mensajes de notificación

Los list box también envían *mensajes de notificación* para informar sobre determinados eventos.

Los mensajes de notificación se reciben a través de un mensaje [WM\\_COMMAND](#). En la palabra de menor peso del parámetro wParam se envía el identificador del control. El manipulador del control se envía en el parámetro lParam y el código del mensaje de notificación en la palabra de mayor peso de wParam.

**Nota:**

En el API de Windows 3.x el código del mensaje de notificación se envía en el parámetro lParam. Hay que tener esto en cuenta si se intenta portar código entre estas plataformas.

Veamos a continuación los mensajes de notificación que existen para los controles list box:

## **Doble clic**

Cada vez que el usuario hace doble clic sobre uno de los ítems de un list box, se envía un mensaje de notificación [LBN\\_DBLCLK](#) a la ventana padre.

## **Falta espacio**

Si no es posible conseguir memoria para completar una operación sobre el list box, se envía un mensaje de notificación [LBN\\_ERRSPACE](#).

## **Pérdida y recuperación de foco**

Cada vez que el usuario selecciona otro control se envía un mensaje de notificación [LBN\\_KILLFOCUS](#).

Cuando el usuario selecciona un control list box, se envía un mensaje de notificación [LBN\\_SETFOCUS](#).

## **Selección y deselección**

Cada vez que la selección de un list box se modifique se envía un mensaje [LBN\\_SELCHANGE](#).

Cuando el usuario cancela la selección de un ítem, se envía el mensaje **LBN\_SELCANCEL**.

```
/* Respuesta a mensaje de notificación de cambio de
selección:
Si la selección cambia se actualiza la ventana. */
case WM_COMMAND:
    switch(LOWORD(wParam)) {
        case ID_LISTA:
            switch(HIWORD(wParam)) {
                case LBN_SELCHANGE:
                    InvalidateRect(hwnd, NULL, TRUE);
                    break;
            }
            break;
```

## Mensajes más comunes

En el [capítulo 8](#) ya vimos algunos de los mensajes de uso más frecuente en list box, los repasaremos ahora y comentaremos algunos más:

Entre los conocidos, tenemos los siguientes mensajes:

**LB\_ADDSTRING** para añadir cadenas a un list box, la dirección de la cadena se envía en el parámetro lParam.

```
sprintf(cad, "NUEVA CADENA");
SendMessage(hwnd, LB_ADDSTRING, 0, (LPARAM)cad);
```

El mensaje **LB\_SELECTSTRING** sirve para seleccionar una cadena determinada. En el parámetro wParam se envía el índice en que debe comenzar la búsqueda y en lParam la dirección de la cadena a buscar.

```
/* Seleccionar primera cadena que empieza por "E",
```

```

después del 6º ítem */
    int i=6;
    ...
    SendMessage(hctrl, LB_SELECTSTRING, (WPARAM)i,
(LPARAM)"E");

```

Para obtener el índice del ítem actualmente seleccionado se usa el mensaje [LB\\_GETCURSEL](#). Este mensaje no precisa parámetros.

```

    int i;
    ...
    i = SendMessage(hctrl, LB_GETCURSEL, 0, 0);

```

Los mensajes [LB\\_GETTEXT](#) y [LB\\_GETTEXTLEN](#) nos sirven para leer cadenas desde un list box. Para el primero se indica en el parámetro wParam el índice del ítem a recuperar, y en lParam la dirección del buffer donde se lee la cadena. El segundo mensaje nos sirve para obtener la longitud de la cadena de un ítem, indicado mediante su índice en el parámetro wParam.

```

/* Obtener cadena seleccionada */
int i, l;
char *cad;
...
i = SendMessage(hctrl, LB_GETCURSEL, 0, 0);
l = SendMessage(hctrl, LB_GETTEXTLEN, (WPARAM)i,
(LPARAM)cad);
cad = (char*)malloc(l+1);
SendMessage(hctrl, LB_GETTEXT, (WPARAM)i, (LPARAM)cad);

```

Pero existen otros mensajes que suelen ser muy útiles a la hora de usar list boxes:

Por ejemplo, podemos eliminar líneas mediante el mensaje [LB\\_DELETESTRING](#), en el que indicaremos en el parámetro wParam el valor del índice a eliminar.

```

/* Eliminar cadena actualmente seleccionada */
int i;
...
i = SendMessage(hctrl, LB_GETCURSEL, 0, 0);
SendMessage(hctrl, LB_DELETESTRING, (WPARAM)i, 0);

```

El mensaje **LB\_FINDSTRING** nos permite buscar una cadena que coincida con el prefijo especificado en el parámetro lParam, a partir del índice indicado en wParam.

```

/* Seleccionar siguiente cadena, a partir de la
seleccionada actualmente, que empiece por "CO" */
int i;
...
i = SendMessage(hctrl, LB_GETCURSEL, 0, 0);
i = SendMessage(hctrl, LB_FINDSTRING, (WPARAM)i,
(LPARAM) "CO");
SendMessage(hctrl, LB_SETCURSEL, (WPARAM)i, 0);

```

El mensaje **LB\_FINDSTRINGEXACT** es parecido, pero no usa el parámetro lParam como un prefijo, sino que busca una cadena que coincida exactamente con ese parámetro.

```

/* Seleccionar la cadena igual a "Portugal" */
int i;
...
i = SendMessage(hctrl, LB_FINDSTRINGEXACT, (WPARAM)-1,
(LPARAM) "Portugal");
SendMessage(hctrl, LB_SETCURSEL, (WPARAM)i, 0);

```

El mensaje **LB\_GETCOUNT** no tiene parámetros, y sirve para obtener el número de elementos que contiene un list box.

```

/* Obtener número de ítems */
int i;

```

```
...
i = SendMessage(hctrl, LB_GETCOUNT, 0, 0);
sprintf(cad, "Número de ítems: %d", i);
MessageBox(hwnd, cad, "List Box", MB_OK);
```

**LB\_GETTOPINDEX** tampoco tiene parámetros, y sirve para recuperar el índice del primer ítem visible de un list box.

```
/* Índice de primer ítem visible */
int i;
...
i = SendMessage(hctrl, LB_GETTOPINDEX, 0, 0);
sprintf(cad, "Pimer ítem visible: %d", i);
MessageBox(hwnd, cad, "List Box", MB_OK);
```

**LB\_INSERTSTRING** nos permite insertar un ítem en una posición determinada por el valor del parámetro wParam, y con el texto indicado en lParam. Esta inserción se hace en la posición indicada, aunque la lista tenga el estilo **LBS\_SORT**.

```
/* Insertar un ítem antes del seleccionado actualmente */
int i;
...
sprintf(cad, "CADENA INSERTADA");
i = SendMessage(hctrl, LB_GETCURSEL, 0, 0);
SendMessage(hctrl, LB_INSERTSTRING, (WPARAM)i,
(LPARAM)cad);
```

Mediante el mensaje **LB\_RESETCONTENT**, sin parámetros, podemos vaciar un list box por completo.

```
/* Vaciar list box */
SendMessage(hctrl, LB_RESETCONTENT, 0, 0);
```

El mensaje `LB_SETCURSEL` nos permite seleccionar un ítem, indicado en el parámetro `wParam`. Además, se elimina la selección previa, y el contenido del list box se desplaza, si es necesario, para mostrar la nueva cadena seleccionada.

```
/* Seleccionar ítem siguiente al actual */
int i;
...
i = SendMessage(hwnd, LB_GETCURSEL, 0, 0);
SendMessage(hwnd, LB_SETCURSEL, (WPARAM)i+1, 0);
```

Si queremos asegurar que un ítem determinado será visible, sin seleccionarlo, podemos usar el mensaje `LB_SETTOPINDEX`, indicando en el parámetro `wParam` el índice del ítem a visualizar.

```
/* Asegurarse de que el ítem 18 es visible */
SendMessage(hwnd, LB_SETTOPINDEX, (WPARAM)18, 0);
```

## Ejemplo 57

### El dato del ítem

En todos los ejemplos que hemos visto siempre hemos recuperado cadenas de un list box, pero también podemos trabajar con índices. Por ejemplo, crearemos un programa que nos muestre las capitales y superficies de varios países, que podremos seleccionar de un list box. Para ello almacenaremos esos datos en un array:

```
struct Pais {
    char *Nombre;
    char *Capital;
    int Superficie;
```

```
} paises[22] =  
{  
    "Argentina", "Buenos Aires", 2766890,  
    "Mexico", "Mexico DC", 1972550,  
    "Brasil", "Brasilia", 8514876,  
    "Peru", "Lima", 1285220,  
    "Colombia", "Bogotá", 1138910,  
    "Bolivia", "La Paz", 1098580,  
    "Venezuela", "Caracas", 912050,  
    "Chile", "Santiago", 756096,  
    "España", "Madrid", 504782,  
    "Paraguay", "Asunción", 406750,  
    "Ecuador", "Quito", 283560,  
    "Uruguay", "Montevideo", 176220,  
    "Nicaragua", "Managua", 129494,  
    "Honduras", "Tegucigalpa", 112090,  
    "Cuba", "La Habana", 110860,  
    "Guatemala", "Guatemala", 108890,  
    "Portugal", "Lisboa", 92391,  
    "Panamá", "Panamá", 78200,  
    "Costa Rica", "San José", 51100,  
    "República Dominicana", "Santo Domingo", 48730,  
    "El Salvador", "San Salvador", 21040,  
    "Puerto Rico", "San Juan", 9104  
};
```

Generalmente trabajaremos con list boxes definidos con el estilo [LBS\\_STANDARD](#), lo cual implica, entre otras cosas, que las cadenas se muestran por orden alfabético.

En nuestro ejemplo esto plantea un problema. El array está ordenado por superficies, no alfabéticamente, por lo tanto, una vez insertadas las cadenas, los índices de los ítems en el list box no coincidirán con los índices en el array, lo cual sería muy útil, ya que podríamos recuperar el índice del ítem activo y mostrar la información correspondiente a ese índice en el array.

Para evitar esto, por supuesto, podemos eliminar el estilo [LBS\\_SORT](#), con lo que el orden del list box coincidiría con el del array. Pero esto no nos interesa, ya que complica la tarea del usuario, que debe buscar en un list box aparentemente desordenado.



Otra forma de evitarlo es leer la cadena seleccionada y buscarla en el array de forma secuencial. Sin embargo, esta solución no es muy elegante, y sería francamente mala si la lista contiene muchos elementos.

También podemos ordenar el array alfabéticamente, pero esta solución tampoco es satisfactoria, ya que podría haber errores de orden, o sencillamente, podríamos usar el mismo array para crear un list box de capitales.

El API nos permite usar otra solución. Ya sabemos que cada ítem tiene asociado un índice y una cadena. Pero también tiene asociado un dato entero de 32 bits: el ítem data, o dato de ítem.

A cada ítem le podemos asignar un valor entero mediante el mensaje `LB_SETITEMDATA`, y recuperarlo mediante `LB_GETITEMDATA`.

Podemos aprovechar que el valor de retorno del mensaje `LB_ADDSTRING` es el índice del ítem insertado, y usar ese valor en el mensaje `LB_SETITEMDATA`, para asignar el valor del índice en el array:

```
void IniciarLista(HWND hctrl)
{
    int i;
    int actual;

    for(i = 0; i < 22; i++) {
        actual = SendMessage(hctrl, LB_ADDSTRING, 0,
(LPARAM)países[i].Nombre);
        SendMessage(hctrl, LB_SETITEMDATA, (LPARAM)actual, i);
    }
}
```

De este modo, podremos recuperar los datos del array correspondientes a un ítem:

```
case WM_PAINT:
```

```

        i = SendMessage(hctrl, LB_GETCURSEL, 0, 0);
        i = SendMessage(hctrl, LB_GETITEMDATA, (WPARAM)i,
0);

        hdc = BeginPaint(hwnd, &ps);
        SetBkMode(hdc, TRANSPARENT);
        sprintf(cad, "País: %s", paises[i].Nombre);
        TextOut(hdc, 300, 20, cad, strlen(cad));
        sprintf(cad, "Capital: %s", paises[i].Capital);
        TextOut(hdc, 300, 40, cad, strlen(cad));
        sprintf(cad, "Superficie: %d Km²",
paises[i].Superficie);
        TextOut(hdc, 300, 60, cad, strlen(cad));
        EndPaint(hwnd, &ps);
        break;

```

Ya veremos que el dato de ítem tiene otras utilidades, pero en muchos casos nos proporciona una forma útil de almacenar un dato relativo a un ítem. Al tratarse de un entero de 32 bits también puede contener punteros.

## Ejemplo 58

### Funciones para ficheros y directorios

Una de las aplicaciones más frecuentes de los list box es la elección de ficheros. Por ese motivo, el API proporciona algunas funciones para iniciar y seleccionar ítems en un list box a partir de los datos de directorios.

La función [DlgDirList](#) nos permite iniciar el contenido de un list box a partir de los ficheros, carpetas, unidades de disco, etc.

Esta función necesita cinco parámetros. El primero es un manipulador de la ventana o diálogo que contiene el list box que vamos a inicializar. El segundo es un puntero a una cadena con el camino del directorio a mostrar. Esta cadena tiene que tener espacio suficiente, ya que la función puede modificar su contenido. El tercer parámetro es el identificador del list box. El cuarto el identificador de un control estático, que se usa para mostrar el camino actualmente

mostrado en el list box. El último parámetro nos permite seleccionar el tipo de entradas que se mostrarán en el list box.

Mediante este último parámetro podemos restringir el tipo de entradas, impidiendo o permitiendo que se muestren directorios o unidades de almacenamiento, o limitando los atributos de los ficheros y directorios a mostrar.

Ya hemos dicho que se necesita un control estático. Como aún no hemos visto el modo de insertar estos controles directamente en la ventana, para este ejemplo lo haremos sin más explicaciones, aunque como se puede ver, no tiene nada de raro:

```
    HWND hestatico;
...
    hestatico = CreateWindowEx(
        0,
        "STATIC",          /* Nombre de la clase */
        "",                /* Texto del título, no tiene
*/
        WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_TABSTOP, /* Estilo */
        9, 4,              /* Posición */
        344, 18,           /* Tamaño */
        hwnd,              /* Ventana padre */
        (HMENU)ID_TITULO, /* Identificador del control
*/
        hInstance,         /* Instancia */
        NULL);             /* Sin datos de creación de
ventana */
    SendMessage(hestatico, WM_SETFONT, (WPARAM)hfont,
MAKELPARAM(TRUE, 0));
```

Por supuesto, podemos usar los comodines '\*' y '?' para los nombres de fichero.

Veamos un ejemplo para uniciar un list box a partir del directorio actual, mostrando los discos y directorios, y limitando los ficheros a los que se ajusten a \*.c:

```

...
    IniciarLista(hwnd, "*.c");
...

void IniciarLista(HWND hwnd, char* p)
{
    char path[512];

    strcpy(path, p);

    DlgDirList(
        hwnd,                /* manipulador de cuadro de diálogo con
list box */
        path,                /* puntero a cadena de camino o nombre
de fichero */
        ID_LISTA,           /* identificador de list box
*/
        ID_TITULO,          /* identificador de control estático
*/
        DDL_DIRECTORY | DDL_DRIVES /* atributos de ficheros a
mostrar */
    );
}

```

La función [DlgDirSelectEx](#) nos permite leer la selección actual de una lista inicializada mediante la función [DlgDirList](#). Si el valor de retorno de esta función es distinto de cero, la selección actual es un directorio o unidad de almacenamiento, por lo que será posible hacer un cambio de directorio. Si el valor de retorno es cero, se trata de un fichero.

Aprovecharemos esto para *navegar* a lo largo de los discos de nuestro ordenador, para lo que responderemos al mensaje de notificación [LBN\\_DBLCLK](#), cambiando a la nueva ubicación o mostrando el nombre del fichero seleccionado:

```

case WM_COMMAND:
    switch(LOWORD(wParam)) {
        case ID_LISTA:
            switch(HIWORD(wParam)) {

```

```

        case LBN_DBLCLK:
            if (DlgDirSelectEx(hwnd, cad, 512,
ID_LISTA)) {
                strcat(cad, "*.c");
                IniciarLista(hwnd, cad);
            } else
            else
                MessageBox(hwnd, cad, "Fichero
seleccionado", MB_OK);
            break;
        ...

```

También existen dos mensajes relacionados con este tema.

El mensaje **LB\_DIR** tiene un uso equivalente a la función **DlgDirList**. En el parámetro wParam se indican los atributos de los ficheros a mostrar, así como si se deben mostrar directorios y unidades de almacenamiento. En el parámetro lParam se suministra el nombre de fichero, (que puede tener comodines) o el camino de los ficheros a insertar.

Por ejemplo, podemos añadir los ficheros de cabecera al contenido del list box, de modo que se muestren los ficheros fuente en c y los de cabecera:

```

void IniciarLista(HWND hwnd, char* p)
{
    char path[512];

    strcpy(path, p);

    DlgDirList(hwnd, path, ID_LISTA, ID_TITULO,
        DDL_DIRECTORY | DDL_DRIVES);

    strcpy(path, "*.h");
    SendMessage(GetDlgItem(hwnd, ID_LISTA), LB_DIR,
        (WPARAM) 0, (LPARAM) path);
}

```

Pero este mensaje no está previsto para usarse junto a la función **DlgDirList**, ya que tienen objetivos parecidos. En su lugar

podemos usar otro mensaje, que nos permite añadir ficheros a una lista previamente creada.

Este otro mensaje, [LB\\_ADDFILE](#), nos permite añadir ficheros sueltos al contenido del list box. Los ficheros sólo se añaden si existen, y también pueden usar comodines. Este ejemplo añadiría los ficheros ejecutables que existan en el directorio actualmente mostrado:

```
...
    SendMessage(GetDlgItem(hwnd, ID_LISTA), LB_ADDFILE,
        0, (LPARAM) "*.exe");
```

## Ejemplo 59

### Listbox de selección sencilla y múltiple

Bien, hasta ahora sólo hemos hablado de list boxes de selección sencilla. En estos list boxes sólo se puede seleccionar un ítem (o ninguno), y cada nueva selección anula a la anterior.

Pero también podemos crear list boxes de selección múltiple, en los que será posible seleccionar rangos de ítems, o varios ítems, aunque no estén seguidos.

Para ello, lo primero será crear el list box con el estilo [LBS\\_MULTIPLESEL](#) o el estilo [LBS\\_EXTENDEDSEL](#).

Ambos estilos definen listas de selección múltiple. La diferencia está en el modo en que se seleccionan los ítems.

En un list box con el estilo [LBS\\_MULTIPLESEL](#) la selección se hace de manera individual, cada vez que se pulsa sobre un ítem no seleccionado, pasa a estado seleccionado, y viceversa.

En un list box con el estilo [LBS\\_EXTENDEDSEL](#), las selecciones se comportan de la misma forma que en un list box de selección sencilla. Pero mediante las teclas de mayúsculas y de control podemos hacer selecciones múltiples. La tecla de mayúsculas nos

permite seleccionar rangos. El último ítem seleccionado se comporta como un "ancla", de modo que si pulsamos la tecla de mayúsculas y pulsamos con el ratón sobre otro ítem se seleccionarán todos los ítems entre en "ancla" y el actual.

Por otra parte, la tecla de control permite usar el list box del mismo modo que si tuviese el estilo `LBS_MULTIPLESEL`. Con la tecla de control pulsada, si pulsamos sobre el ratón en un ítem seleccionado, se deseleccionará, y viceversa.

## Selecciones

Disponemos de varios mensajes para tratar la selección de ítems en list boxes de selección múltiple.

El mensaje `LB_GETSELCOUNT` nos sirve para obtener el número de ítems seleccionados actualmente en un list box:

```
int nSeleccionados;
...
nSeleccionados = SendMessage(hlista, LB_GETSELCOUNT, 0,
0);
```

Mediante el mensaje `LB_GETSELITEMS` podemos obtener una lista de los índices de los ítems seleccionados. Para ello indicaremos en el parámetro `lParam` un puntero a un array de enteros, en los que recibiremos la lista de índices, y en `wParam` el número máximo de ítems que podemos recuperar:

```
int nSeleccionados;
int *seleccionado;
...
nSeleccionados = SendMessage(hlista, LB_GETSELCOUNT, 0,
0);
/* Obtener memoria para nSeleccionados índices */
seleccionado = (int *)malloc(nSeleccionados*sizeof(int));
/* Obtener la lista */
SendMessage(hlista, LB_GETSELITEMS,
(WPARAM)nSeleccionados, (LPARAM)seleccionado);
```

```

/* Tratamiento */
for(i = nSeleccionados-1; i >= 0; i--) {
    SendMessage(hlista, LB_GETTEXT,
(WPARAM)seleccionado[i], (LPARAM)cad);
}
/* Liberar buffer */
free(seleccionado);

```

Otra alternativa para averiguar qué ítems están seleccionados consiste en enviar un mensaje **LB\_GETSEL** a cada uno de los ítems. Este mensaje nos devuelve un valor distinto de cero si el ítem está seleccionado, y cero si no lo está. En el parámetro wParam indicaremos el índice del ítem:

```

int n, i;
char cad[64];
...
n = SendMessage(hlista, LB_GETCOUNT, 0, 0);
for(i = 0; i < n; i++) {
    if(SendMessage(hlista, LB_GETSEL, (WPARAM)i, 0)) {
        SendMessage(hlista, LB_GETTEXT, (WPARAM)i,
(LPARAM)cad);
        MessageBox(hwnd, cad, "Ítem seleccionado", MB_OK);
    }
}

```

Finalmente, el mensaje **LB\_SETSEL** nos permite seleccionar o deselectionar un ítem determinado. En el parámetro lParam indicaremos el índice del ítem, en el parámetro wParam especificaremos mediante el valor **TRUE** que queremos seleccionar el ítem, y mediante **FALSE** que queremos deselectionarlo:

```

int n, i;
char cad[64];
...
/* Seleccionar ítems de posiciones pares (con índice
impar) */
n = SendMessage(hlista1, LB_GETCOUNT, 0, 0);
for(i = 0; i < n; i++) {

```



```
SendMessage(hlista1, LB_SETSEL, (WPARAM)i%2,  
(LPARAM)i);  
}
```

También podemos seleccionar rangos de ítems mediante el mensaje **LB\_SELITEMRANGE**. En este caso, en el parámetro wParam también indicaremos el tipo de selección, **TRUE** para seleccionar y **FALSE** para deselectionar. En el parámetro lParam indicaremos el rango, en la palabra de menor peso el primer ítem y en la de mayor peso, el último. Para crear un **LPARAM** usaremos la macro **MAKELPARAM**:

```
int n;  
...  
n = SendMessage(hlista1, LB_GETCOUNT, 0, 0);  
SendMessage(hlista1, LB_SELITEMRANGE, (WPARAM)TRUE,  
MAKELPARAM(0, n));
```

El mensaje **LB\_SELITEMRANGEEX** es similar, aunque en el parámetro wParam se indica el primer ítem y en el parámetro lParam el último. Si el índice del primero es menor que el del segundo, se seleccionará el rango. Si el mayor es el segundo, el rango se deselectionará.

## Mensajes especiales para list box de selección extendida

Existen ciertos mensajes para manipular el ítem ancla y el ítem actual (caret).

El mensaje **LB\_GETANCHORINDEX** nos permite obtener el ítem ancla, y el mensaje **LB\_GETCARETINDEX**, el ítem actual. En ninguno de los dos mensajes se necesitan parámetros:

```
int i;
```

```

    char cad[64];
    ...
    case CM_ANCLA:
        i = SendMessage(hlista2, LB_GETANCHORINDEX, 0, 0);
        SendMessage(hlista2, LB_GETTEXT, (WPARAM)i,
(LPARAM)cad);
        MessageBox(hwnd, cad, "Ítem ancla", MB_OK);
        break;
    case CM_CARET:
        i = SendMessage(hlista2, LB_GETCARETINDEX, 0, 0);
        SendMessage(hlista2, LB_GETTEXT, (WPARAM)i,
(LPARAM)cad);
        MessageBox(hwnd, cad, "Ítem caret", MB_OK);
        break;
    ...

```

De forma simétrica, podemos asignar los ítems ancla y actual mediante los mensajes [LB\\_SETANCHORINDEX](#) y [LB\\_SETCARETINDEX](#), respectivamente. En ambos casos indicaremos en el parámetro wParam el valor del índice en cuestión, (aunque ignoro qué utilidad puede tener esto):

```

    int n;
    ...
    n = SendMessage(hlista2, LB_GETCOUNT, 0, 0);
    SendMessage(hlista2, LB_SETANCHORINDEX, 0, 0);
    SendMessage(hlista2, LB_SETCARETINDEX, (WPARAM)n, 0);

```

## Ejemplo 60

### List box sin selección

Además de los list box de selección sencilla y de selección múltiple, también es posible crear list boxes sin selección. Para ello bastará con crear el list box con el estilo [LBS\\_NOSEL](#).

Podemos usar estos list boxes para mostrar listas de valores para una consulta por parte del usuario, pero que no precisen una

selección.

## List box multicolumna

Los ítems en un list box no tienen por qué mostrarse en una única columna, como hemos hecho en los ejemplos anteriores. Si se especifica el estilo `LBS_MULTICOLUMN` se aprovechará toda la anchura del list box para mostrar varias columnas de ítems.

En los list boxes de varias columnas se aprovecha toda la altura del control para mostrar tantos ítems como sea posible, y el resto se muestran en otras columnas. Se añadirán tantas columnas como sea necesario, y será posible desplazarse horizontalmente, aunque para poder usar la barra de desplazamiento horizontal habrá que especificar el estilo `WS_HSCROLL` al crear el control. Las columnas que no quepan en el área del list box no serán visibles pero al desplazarnos lateralmente se mostrarán nuevas columnas y se ocultarán por el lado contrario.

Disponemos de un mensaje para establecer la anchura de las columnas, se trata de `LB_SETCOLUMNWIDTH`. En el parámetro `WParam` indicaremos la anchura de las columnas, en pixels:

```
SendMessage(hctrl, LB_SETCOLUMNWIDTH, (WPARAM) 40, 0);
```

## Ejemplo 61

### Paradas de tabulación

En principio no es posible crear list boxes en los que a cada fila corresponda un ítem, y que para cada ítem se creen varias columnas con informaciones diferentes. (Esto es algo que se hace con otro tipo de control que veremos más adelante: los list view).

Sin embargo, existe una forma limitada de hacer algo parecido. Consiste en usar el estilo [LBS\\_USETABSTOPS](#), y en separar las columnas dentro de cada ítem con caracteres de tabulación. Si no se especifica este estilo, los caracteres de tabulación no se expanden en espacios, y el list box no tendrá el aspecto de una tabla.

Podemos definir la anchura de cada columna mediante el mensaje [LB\\_SETTABSTOPS](#), indicando en el parámetro wParam el número de paradas de tabulación y en el parámetro lParam un array con las separaciones de cada parada en pixels:

```
int tab[4] = {50,80,130,160};
int i;
char cad[128];
...
for(i = 0; i < 40; i++) {
    sprintf(cad, "Ítem %03d\tcol 2\tcolumna 3\tinfo
x\t%d", i, i*213);
    SendMessage(hlista, LB_ADDSTRING, 0, (LPARAM)cad);
}
...
SendMessage(hctrl, LB_SETTABSTOPS, (WPARAM)4,
(LPARAM)tab);
```

## Ejemplo 62

### Actualizaciones de gran número de ítems

Hay dos posibles situaciones de potencialmente peligrosas en las las actualizaciones que afecten a muchos ítems en un list box.

Por una parte, el proceso puede requerir una cantidad importante de memoria, cuando se añaden muchos ítems.

Por otra parte, el proceso puede requerir mucho tiempo, ya sea porque se deben añadir muchos ítems o porque se deben hacer

muchas modificaciones que impliquen el borrado e inserción de ítems.

## Optimizar la memoria

En versiones de Windows anteriores al uso de la memoria virtual, era necesario tener en cuenta la memoria disponible antes de insertar un gran número de ítems en un list box. Para eso se usaba el mensaje `LB_INITSTORAGE`, en el que indicamos en el parámetro `wParam` el número de ítems a añadir, y en el parámetro `lParam` la cantidad de memoria estimada necesaria para acomodar esos ítems.

```
/* Prepararse para insertar 10000 ítems de
32 bytes por ítem, aproximadamente */
SendMessage(hctrl, LB_INITSTORAGE, 10000, 320000);
IniciarLista(hctrl);
```

No es necesario ser demasiado preciso con la cantidad de memoria requerida, se trata sólo de una estimación, si nos quedamos cortos, los ítems que no quepan se insertarán del modo normal. Si nos quedamos largos, la memoria sobrante se podrá aprovechar en nuevas inserciones.

Este mensaje sólo es necesario en Windows 95, en NT no nos preocupa la memoria necesaria para almacenar los ítems, ya que el modelo de memoria virtual dispone de una cantidad prácticamente ilimitada.

## Optimizar el tiempo

El problema del tiempo sí es importante. Cada vez que se añade o elimina un ítem, el list box intenta actualizar la pantalla para reflejar los cambios. Esto, cuando los cambios son muy numerosos,

hará que aparentemente la aplicación no responda, y que el tiempo invertido en las actualizaciones sea mayor del necesario.

Para evitar esto podemos hacer uso del estilo `LBS_NOREDRAW`. Si este estilo está activo no se actualizará la ventana del list box aunque se produzcan cambios. Este estilo se puede activar y desactivar mediante un mensaje `WM_SETREDRAW`, indicando en el parámetro `wParam` el valor `TRUE` para desactivarlo (activar el redibujado) o `FALSE` para activarlo (desactivar el redibujado).

```
SendMessage(hctrl, WM_SETREDRAW, FALSE, 0);  
IniciarLista(hctrl);  
SendMessage(hctrl, WM_SETREDRAW, TRUE, 0);
```

## Ejemplo 63

### Responder al teclado

Podemos hacer que nuestro control list box responda a determinadas pulsaciones del teclado, cuando tenga el foco, usando el estilo `LBS_WANTKEYBOARDINPUT`.

Mediante este estilo, la ventana padre del list box recibirá un mensaje `WM_VKEYTOITEM` cada vez que el usuario pulse una tecla.

Nuestro procedimiento de ventana o diálogo podrá procesar este mensaje y actuar en consecuencia. En el parámetro `wParam` recibiremos dos valores, en la palabra de menor peso el código de tecla virtual de la tecla pulsada, y en la palabra de mayor peso, la posición del caret. En el parámetro `lParam` recibiremos el manipulador del control list box.

En el siguiente ejemplo podemos hacer que nuestro list box responda a la tecla 'B' borrando el ítem actualmente seleccionado, y a la tecla 'I' insertando de nuevo los valores iniciales:

```

        case WM_VKEYTOITEM:
            if((HWND)lParam == hctrl) { /* Asegurarse de que
el mensaje proviene de nuestra lista */
                switch(LOWORD(wParam)) {
                    case 'B': /* Borrar actual */
                        i = SendMessage(hctrl, LB_GETCURSEL, 0, 0);
                        SendMessage(hctrl, LB_DELETESTRING,
(WPARAM)i, 0);
                        SetFocus(hctrl);
                        return -2; /* Tratamiento de tecla
terminado */
                    case 'I': /* Leer valores iniciales */
                        IniciarLista(hctrl);
                        SetFocus(hctrl);
                        return -2; /* Tratamiento de tecla
terminado */
                }
            }
            return -1; /* Acción por defecto */

```

El valor de retorno es importante. Un valor de -1 indica que el list box debe realizar la acción por defecto definida para la tecla. Esto permite, en nuestro ejemplo, que las teclas del cursor (y en general, todas menos la 'B' y la 'I'), sigan realizando sus funciones normales. Un valor de -2 indica que el list box no tiene que realizar ninguna acción para esta pulsación de tecla. Un valor igual o mayor que 0 se refiere a un índice de un ítem, e indica que el list box debe realizar la acción por defecto con ese índice.

## Ejemplo 64

### Aspectos gráficos del list box

En cuanto al aspecto gráfico del list box tenemos otras opciones que podemos controlar.

#### Ajustar la anchura de un list box

Por una parte, ya vimos que podemos añadir una barra de desplazamiento horizontal creando nuestro list box con el estilo [WS\\_HSCROLL](#). Esto lo podemos hacer aunque no se trate de un list box de columnas múltiples. Puede ser útil si la anchura de los ítems sobrepasa la del list box.

Sin embargo, usar este estilo no asegura que la barra de desplazamiento sea mostrada. Para que la barra aparezca hay que ajustar la extensión horizontal del list box mediante un mensaje [LB\\_SETHORIZONTALEXTENT](#), indicando en el parámetro wParam la nueva extensión horizontal, en pixels.

Si la extensión horizontal es mayor que la anchura del list box, se mostrará la barra de desplazamiento, en caso contrario la barra no aparecerá.

Esto nos plantea una duda, ¿cómo calcular la extensión necesaria según las longitudes de las cadenas contenidas en el list box?

Bueno, podríamos hacerlo *a ojo*, pero esta técnica es arriesgada, ya que si nos quedamos cortos no será posible visualizar por completo algunos ítems.

Lo mejor es calcular la longitud de cada cadena al insertarla, y si es mayor que la extensión actual, actualizar el valor de la extensión. Para obtener el valor de la extensión actual se usa el mensaje [LB\\_GETHORIZONTALEXTENT](#).

Claro que esto plantea un problema si se eliminan ítems, ya que nos obligaría a calcular las longitudes de todas las cadenas que quedan en el list box. Sin embargo, podemos ignorar estos casos, y mantener la extensión, ya que la visibilidad de todos los ítems está asegurada.

Para calcular la longitud de una cadena en pixes, vimos en el [capítulo 24](#), que podemos usar la función [GetTextExtentPoint32](#), por ejemplo, en la siguiente función:

```
int CalculaLongitud(HWND hwnd, char *cad)
{
```



```

HDC hdc;
SIZE tam;
HFONT hfont;

hfont = (HFONT)GetStockObject( DEFAULT_GUI_FONT );
hdc = GetDC(hwnd);
SelectObject(hdc, hfont);
GetTextExtentPoint32(hdc, cad, strlen(cad), &tam);
/*LPtoDP(hdc, (POINT *)&tam, 1);*/
ReleaseDC(hwnd, hdc);

return tam.cx;
}

```

Para que el cálculo sea correcto debemos seleccionar en el DC la misma fuente que usamos en el list box. Además, habría que tener en cuenta que la función [GetTextExtentPoint32](#) devuelve el tamaño de la cadena en unidades lógicas, y en rigor habría que convertir esos valores a unidades de dispositivo. Esto es innecesario, ya que en un control no se realiza ninguna proyección.

Así, cada vez que insertemos un ítem en el list box, deberemos comprobar si resulta ser el más largo:

```

char item[300];
int x;
int eActual;

eActual = SendMessage(hlista, LB_GETHORIZONTALEXTENT, 0, 0);

strcpy(item, "Ítem de una anchura tal que no cabe en "
           "el list box que hemos definido, o al menos no "
           "debería caber, "
           "si las cosas salen tal y como las hemos calculado, "
           "claro.");

x = CalculaLongitud(hlista, cad);
if(x > eActual) eActual = x;
SendMessage(hlista, LB_ADDSTRING, 0, (LPARAM)cad);
SendMessage(hlista, LB_SETHORIZONTALEXTENT, eActual, 0);

```

## Ajustar la altura de los ítems

Por defecto, la altura de los ítems se calcula en función de la fuente asignada al list box. Podemos obtener el valor de la altura del ítem mediante el mensaje [LB\\_GETITEMHEIGHT](#). Si se trata de un list box con un estilo *owner-draw* cada ítem puede tener una altura diferente, y se puede especificar el índice del ítem en el parámetro wParam. En los list box normales, el valor de wParam debe ser cero.

```
h = SendMessage(hctrl, LB_GETITEMHEIGHT, 0, 0);
```

Para modificar la altura de un ítem se usa el mensaje [LB\\_SETITEMHEIGHT](#), en el caso de list boxes con un estilo *owner-draw* se puede asignar una altura diferente a cada ítem. En ese caso, especificaremos el índice del ítem en el parámetro wParam, y la altura deseada en la palabra de menor peso del parámetro lParam, usando la macro [MAKELPARAM](#). Veremos esto con más detalle al estudiar los estilos *owner-draw*.

```
SendMessage(hctrl, LB_SETITEMHEIGHT, 0,  
MAKELPARAM(30,0));  
InvalidateRect(hctrl, NULL, TRUE);
```

## Ítems y coordenadas

Podemos obtener las coordenadas del rectángulo que contiene a un ítem determinado mediante el mensaje [LB\\_GETITEMRECT](#), indicando en el parámetro wParam el índice del ítem y en el parámetro lParam un puntero a una estructura [RECT](#) que recibirá las coordenadas del rectángulo:

```
int i;
RECT re;
...
i = SendMessage(hctrl, LB_GETCURSEL, 0, 0);
SendMessage(hctrl, LB_GETITEMRECT, i, (LPARAM)&re);
```

También podemos obtener el índice del ítem correspondiente a las coordenadas de un punto dentro del list box. Para ello usaremos el mensaje `LB_ITEMFROMPOINT`, indicando en el parámetro `lParam` las coordenadas del punto, en la palabra de menor peso la coordenada `x` y en la de mayor peso, la coordenada `y`. Usaremos la macro `MAKELPARAM` para crear el valor del parámetro a partir de las coordenadas:

```
int i;
...
i = SendMessage(hctrl, LB_ITEMFROMPOINT, 0,
MAKELPARAM(40, 123));
```

## Ejemplo 65

### Localizaciones

Ya hemos visto que en los controles list box los ítems se muestran por orden alfabético, al menos en los que hemos usado hasta ahora. Pero el orden alfabético no es algo universal, y puede cambiar dependiendo del idioma.

Generalmente esto no nos preocupará, ya que el idioma usado para elegir el orden se toma del propio sistema. Sin embargo, puede haber casos en que nos interese modificar o conocer el idioma usado en un list box.

Para obtener el valor de la localización actual se usa el mensaje `LB_GETLOCALE`. El valor de retorno es un entero de 32 bits, en el

que la palabra de menor peso contiene el código de país, y el de mayor peso el del lenguaje, este último a su vez, se compone de un identificador de lenguaje primario y un identificador de sublenguaje.

Se pueden usar las macros [PRIMARYLANGID](#) y [SUBLANGID](#) para obtener el identificador de lenguaje primario y el de sublenguaje, respectivamente.

```
int i;
char cad[120];
...
i = SendMessage(hctrl, LB_GETLOCALE, 0, 0);
sprintf(cad, "País %d, id lenguaje primario %d, "
        "id de sublenguaje %d",
        HIWORD(i), PRIMARYLANGID(LOWORD(i)),
        SUBLANGID(LOWORD(i)));
MessageBox(hwnd, cad, "Localización", MB_OK);
```

También podemos modificar la localización actual mediante un mensaje [LB\\_SETLOCALE](#), indicando en el parámetro wParam el nuevo valor de localización. Podemos crear uno de estos valores mediante las macros [MAKELCID](#) y [MAKELANGID](#):

```
SendMessage(hctrl, LB_SETLOCALE,
            MAKELCID(MAKELANGID(LANG_SPANISH,
                                SUBLANG_SPANISH),
                    SSORT_DEFAULT), 0);
```

La macro [MAKELCID](#) crea un identificador de localización a partir de un identificador de lenguaje y una constante que debe ser [SSORT\\_DEFAULT](#).

La macro [MAKELANGID](#) crea un identificador de lenguaje a partir de un identificador de lenguaje primario y de un identificador de sublenguaje.

## Ejemplo 66

## Otros estilos

Nos quedan algunas cosas que comentar sobre los estilos de los controles list box.

Generalmente, hasta ahora, hemos basado nuestros list boxes en el estilo `LBS_STANDARD`. En realidad, este estilo se define como la combinación de dos estilos: `LBS_SORT` y `LBS_NOTIFY`.

El estilo `LBS_SORT` indica que los ítems en el list box deben mostrarse ordenados alfabéticamente. El estilo `LBS_NOTIFY` indica que se debe enviar un mensaje de notificación a la ventana padre del control cada vez que el usuario haga clic o boble clic sobre un ítem.

Si no se especifica el estilo `LBS_SORT`, los ítems se muestran en el mismo orden en que se añaden, (excepto aquellos insertados mediante el mensaje `LB_INSERTSTRING`).

Si no se especifica el estilo `LBS_NOTIFY`, la ventana padre no recibirá los mensajes de notificación: `LBN_DBLCLK`, `LBN_SELCHANGE` y `LBN_SELCANCEL`.

Además, aunque no se especifique, todos los controles list box que no tengan un estilo *owner-draw*, se definen con el estilo `LBS_HASSTRINGS` por defecto. Este estilo indica que los ítems consisten en cadenas de texto, el sistema se encarga de mantener la memoria para almacenar estas cadenas.

También podemos usar el estilo `LBS_DISABLENOSCROLL`, de modo que la barra de desplazamiento vertical se muestre siempre, aunque todos los ítems puedan ser visualizados. Este estilo no se puede aplicar a los controles con el estilo `LBS_MULTICOLUMN`.

Por último, hay que comentar algo sobre las dimensiones verticales de los controles list box. Cuando se crea uno de estos controles, la altura del control no se respeta de forma exacta, sino que se reduce lo necesario para que el list box contenga un número exacto de ítems. Así, si por ejemplo, indicamos una altura de 184, y cada ítem ocupa 18, la altura usada para crear el control será de 180, ya que en la distancia especificada caben 18 ítems, pero no 19.

De este modo no se mostrará una parte de un ítem, y el control se dibujará más rápidamente.

Este comportamiento se puede evitar mediante el uso del estilo [LBS\\_NOINTEGRALHEIGHT](#). Los list boxes con este estilo se crearán con la altura exacta indicada al definirlos.

## List box a medida (owner-draw)

Al crear cualquier control casi siempre se puede especificar el estilo *owner-draw*, que quiere decir que la ventana propietaria del control es la responsable del trazado gráfico del control, en lugar de ser el propio procedimiento de ventana del control en que se encargue de esa tarea, como hemos hecho hasta ahora en todos los casos.

Esto deja la responsabilidad de todos los aspectos gráficos a nuestra aplicación, y en concreto, a nuestro procedimiento de ventana o diálogo.

En realidad, actuar de este modo nos complica la vida, pero a cambio, nos da mucho mayor control sobre el aspecto gráfico de las ventanas, y si nos interesa personalizar nuestras aplicaciones, tendremos que recurrir a este estilo.

## Estilos owner-draw para list box

Existen dos estilos distintos *owner-draw* que se pueden aplicar a los controles list box [LBS\\_OWNERDRAWFIXED](#) y [LBS\\_OWNERDRAWVARIABLE](#).

El primero define controles list box owner-draw en los que la altura de todos los ítems es la misma. En el segundo caso, las alturas de cada ítem pueden ser diferentes.

Recordarás que en el punto anterior comentamos que los controles list box con estilos owner-draw no tienen activado por defecto el estilo [LBS\\_HASSTRINGS](#).

Esto es, en cierta medida, bastante lógico, ya que intentamos personalizar el aspecto del control, por lo que es probable que éste no contenga cadenas, o al menos, no sólo cadenas.

Sin embargo, es posible que aún tratándose de un list box con un estilo owner-draw, nuestro control contenga cadenas. En ese caso podemos activar el estilo [LBS\\_HASSTRINGS](#), sobre todo si queremos que los ítems se muestren por orden alfabético.

```
hctrl = CreateWindowEx(
    0,
    "LISTBOX",          /* Nombre de la clase */
    "",                 /* Texto del título */
    LBS_HASSTRINGS | LBS_STANDARD |
LBS_OWNERDRAWVARIABLE |
    WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_TABSTOP, /* Estilo */
    9, 19,             /* Posición */
    320, 250,          /* Tamaño */
    hwnd,              /* Ventana padre */
    (HMENU)ID_LISTA,   /* Identificador del control
*/
    hInstance,         /* Instancia */
    NULL);             /* Sin datos de creación de
ventana */
```

Si no activamos el estilo [LBS\\_HASSTRINGS](#), el valor que usemos al insertar el ítem será almacenado en el dato del ítem de 32 bits.

```
void IniciarLista(HWND hctrl)
{
    int i;

    for(i = 0; i < 22; i++)
        SendMessage(hctrl, LB_ADDSTRING, 0, i);
}
```

## List box owner-draw de altura fija

La ventana propietaria del control recibirá el mensaje `WM_MEASUREITEM` cuando el control list box sea creado.

En el parámetro `lParam` recibiremos un puntero a una estructura `MEASUREITEMSTRUCT` que contiene las dimensiones del control list box.

En el parámetro `wParam` recibiremos el valor del identificador del control, o lo que es lo mismo, el valor del miembro `CtlID` de la estructura `MEASUREITEMSTRUCT` apuntada por el parámetro `lParam`. Este valor identifica el control del que procede el mensaje `WM_MEASUREITEM`.

Tengamos en cuenta que pueden existir varios controles con el estilo `owner-draw`, y no tienen por qué ser necesariamente del tipo `list-box`. Si este valor es cero, el mensaje fue enviado por un menú. Si el valor es distinto de cero, el mensaje fue enviado por un combobox o por un listbox.

Nuestra aplicación debe rellenar de forma adecuada la estructura `MEASUREITEMSTRUCT` apuntada por el parámetro `lParam` regresar. De este modo se indica al sistema operativo qué dimensiones tiene el control.

El mensaje `WM_MEASUREITEM` se envía a la ventana propietaria del list box antes de enviar el mensaje `WM_INITDIALOG` o `WM_CREATE`, de modo que en ese momento Windows aún no ha determinado la altura y anchura de la fuente usada en el control.

Si se procesa este mensaje se debe retornar el valor `TRUE`.

```
switch(msg)                                /* manipulador del mensaje
*/
{
    case WM_CREATE:
        ...
    case WM_MEASUREITEM:
        lpmis = (LPMEASUREITEMSTRUCT) lParam;
        lpmis->itemHeight = 20;
        return TRUE;
    ...
}
```



## List box owner-draw de altura variable

En este caso, la ventana propietaria del control recibirá el mensaje `WM_MEASUREITEM` cada vez que se inserte un nuevo ítem en el control list box. Esto nos permitirá ajustar la altura de cada ítem con valores diferentes.

El proceso del mensaje es idéntico que con el estilo `LBS_OWNERDRAWFIXED`. La diferencia es que este mensaje se enviará para cada ítem, y siempre después del mensaje `WM_INITDIALOG` o `WM_CREATE`.

## Dibujar cada ítem

Tanto en un caso como en el otro, Windows enviará un mensaje cada vez que se inserte un nuevo ítem, cuando el estado de un ítem cambie o cuando un ítem deba ser mostrado.

Esto se hace mediante un mensaje `WM_DRAWITEM`. En el parámetro `wParam` recibiremos el identificador del control del que procede el mensaje, o cero si es un menú. En el parámetro `lParam` recibiremos un puntero a una estructura `DRAWITEMSTRUCT`, que contiene toda la información relativa al ítem que hay que mostrar.

Si se procesa este mensaje hay que retornar el valor `TRUE`.

Procesar este mensaje puede ser un proceso bastante complejo, ya que el estado de un ítem puede tomar varios valores diferentes, y seguramente, cuando decidimos crear un control owner-draw es porque queremos hacer algo especial.

La estructura `DRAWITEMSTRUCT` tiene esta forma:

```
typedef struct tagDRAWITEMSTRUCT {    // dis
    UINT    CtlType;
    UINT    CtlID;
    UINT    itemID;
    UINT    itemAction;
    UINT    itemState;
    HWND    hwndItem;
```

```
HDC    hDC;  
RECT   rcItem;  
DWORD  itemData;  
} DRAWITEMSTRUCT;
```

En nuestro caso, *CtlType* tendrá el valor ODT\_LISTBOX, pero tengamos en cuenta que habrá que discriminar este miembro si tenemos controles owner-draw de distintos tipos.

*CtlID* contiene el identificador del control, igual que el parámetro *wParam*.

*itemID* contiene el índice del ítem . Si el list box está vacío, el valor será -1.

*itemAction* puede tener tres valores diferentes, que en ocasiones requerirán un tratamiento distinto por parte de nuestro programa:

- **ODA\_DRAWENTIRE** indica que el ítem debe ser dibujado por entero.
- **ODA\_FOCUS** indica que el control ha perdido o recuperado el foco. Para saber si se trata de uno u otro caso se debe comprobar el miembro *itemState*.
- **ODA\_SELECT** indica que el estado de selección del ítem ha cambiado. Para saber si el ítem está ahora seleccionado o no también se debe comprobar el miembro *itemState*.

*itemState* indica el estado del ítem. El valor puede ser uno o una combinación de los siguientes:

- **ODS\_DEFAULT** se trata del ítem por defecto.
- **ODS\_DISABLED** el ítem está deshabilitado.
- **ODS\_FOCUS** el ítem tiene el foco.
- **ODS\_SELECTED** el ítem está seleccionado.

*hwndItem* contiene el manipulador de ventana del control.

*hDC* contiene el manipulador de contexto de dispositivo del control. Este valor nos será muy útil, ya que el proceso de este mensaje será en encargado de dibujar el ítem.

*rcItem* contiene un rectángulo que define el contorno del ítem que estamos dibujando. Además este rectángulo define una región de recorte, de modo que no podremos dibujar nada fuera de él.

*itemData* contiene el valor del 32 bits asociado al ítem.

Con esto tenemos toda la información necesaria para dibujar cada ítem, y nuestro programa será el responsable de diferenciar los distintos estados de cada uno.

```
case WM_DRAWITEM:
    lpdis = (LPDRAWITEMSTRUCT) lParam;
    if(lpdis->itemID == -1) { /* Se trata de un
menú, no hacer nada */
        break;
    }
    switch (lpdis->itemAction) {
        case ODA_SELECT:
        case ODA_DRAWENTIRE:
        case ODA_FOCUS:
            /* Borrar el contenido previo */
            FillRect(lpdis->hDC, &lpdis->rcItem, (HBRUSH)
(COLOR_WINDOW+1));
            /* Obtener datos de las medidas de la fuente
*/
            GetTextMetrics(lpdis->hDC, &tm);
            /* Calcular la coordenada y para escribir el
texto de ítem */
            y = (lpdis->rcItem.bottom + lpdis->rcItem.top
- tm.tmHeight) / 2;
            /* Los países cuya superficie sea mayor que
92391 km2 se muestran en verde,
el resto, en azul */
            if(países[lpdis->itemData].Superficie >
92391)
                SetTextColor(lpdis->hDC, RGB(0,128,0));
            else
                SetTextColor(lpdis->hDC, RGB(0,0,255));
            /* Mostrar el texto */
            TextOut(lpdis->hDC, 6, y,
                países[lpdis->itemData].Nombre,
                strlen(países[lpdis->itemData].Nombre));
            /* Si el ítem está seleccionado, trazar un
rectángulo negro alrededor */
            if (lpdis->itemState & ODS_SELECTED) {
```

```
        SetTextColor(lpdis->hDC, RGB(0,0,0));  
        DrawFocusRect(lpdis->hDC, &lpdis->rcItem);  
    }  
}  
break;  
...
```

Este ejemplo usa la lista de países de ejemplos anteriores, hemos hecho que los países de más de 92391 km<sup>2</sup> se muestren en color verde, y el resto en azul.

Por supuesto, esta es una aplicación muy sencilla de un list box owner-draw. Es posible personalizar tanto como queramos estos controles, mostrando mapas de bits o cualquier gráfico que queramos.

También hemos hecho uso de una función nueva: [DrawFocusRect](#). Esta función sirve para trazar un rectángulo que indique que el ítem tiene el foco. Este rectángulo se traza usando el modo XOR, por lo que dos llamadas consecutivas para el mismo rectángulo eliminan la marca.

## El mensaje WM\_DELETEITEM

Cuando se elimina un ítem de un list box cuyo dato de ítem no sea nulo, en Windows 95; o para ítems pertenecientes a controles owner draw, en el caso de Windows NT , el sistema envía un mensaje [WM\\_DELETEITEM](#) al procedimiento de ventana de la ventana propietaria del control. Concretamente, esto ocurre cuando se usan los mensajes [LB\\_DELETESTRING](#) o [LB\\_RESETCONTENT](#) o cuando el propio control es destruido.

Esto nos da una oportunidad de tomar ciertas decisiones o realizar ciertas tareas cuando algunos ítems concretos son eliminados.

En el parámetro wParam recibiremos el identificador del control en el que se ha eliminado el ítem. En el parámetro lParam recibiremos un puntero a una estructura [DELETEITEMSTRUCT](#). Esta estructura está definida como:

```
typedef struct tagDELETEITEMSTRUCT { // ditms
    UINT CtlType;
    UINT CtlID;
    UINT itemID;
    HWND hwndItem;
    UINT itemData;
} DELETEITEMSTRUCT;
```

*CtlType* contiene el valor ODT\_LISTBOX.

*CtlID* contiene el valor del identificador del control.

*itemID* el valor del índice del ítem eliminado.

*hwndItem* el manipulador de ventana del control.

*itemData* el dato del ítem asignado al ítem eliminado.

## Ejemplo 67

### Otros mensajes para list box con estilos owner-draw

Disponemos de otros mensajes destinados a controles owner-draw.

El mensaje [LB\\_GETITEMHEIGHT](#) se puede usar para obtener la altura de los ítems en un list box owner-draw. Si el control tiene el estilo [LBS\\_OWNERDRAWFIXED](#) tanto el parámetro lParam como wParam deben ser cero. Si el control tiene el estilo [LBS\\_OWNERDRAWVARIABLE](#), el parámetro wParam debe contener el índice del ítem cuya altura queramos recuperar.

De forma simétrica, disponemos del mensaje [LB\\_SETITEMHEIGHT](#) para ajustar la altura de los ítems. Si se trata de un control con el estilo [LBS\\_OWNERDRAWFIXED](#) debe indicarse cero para el parámetro wParam, y la altura se especifica en el parámetro lParam, para lo que será necesario usar la macro [MAKELPARAM](#):

```
SendMessage(hctrl, LB_SETITEMHEIGHT, 0, MAKELPARAM(23, 0));
```

Si se trata de un control con el estilo [LBS\\_OWNERDRAWVARIABLE](#) procederemos del mismo modo, pero indicando en el parámetro wParam el índice del ítem cuya altura queremos modificar.

## Definición del orden

Por último, cuando un control list box tiene el estilo [LBS\\_SORT](#), el procedimiento de ventana de la ventana propietaria del control recibe uno o varios mensajes [WM\\_COMPAREITEM](#) para determinar la posición de cada nuevo ítem insertado en el control.

Esto nos permite definir nuestro propio orden para los ítems en el control, en lugar de usar el orden alfabético por defecto.

El mensaje se puede recibir varias veces para cada ítem insertado, ya que generalmente no será suficiente una comparación para determinar el orden.

En el parámetro wParam recibiremos el identificador del control, y en lParam un puntero a una estructura [COMPAREITEMSTRUCT](#), con todos los datos necesarios para determinar el orden entre dos ítems del list box. Esta estructura tiene esta definición:

```
typedef struct tagCOMPAREITEMSTRUCT { // cis
    UINT CtlType;
    UINT CtlID;
    HWND hwndItem;
    UINT itemID1;
    DWORD itemData1;
    UINT itemID2;
    DWORD itemData2;
} COMPAREITEMSTRUCT;
```

*CtlType* contendrá el valor `ODT_LISTBOX`.

*CtlID* el valor del identificador del control.

*hwndItem* el manipulador de ventana del control.

*itemID1* el índice del primer ítem a comparar.

*itemData1* el valor del dato del ítem del primer ítem a comparar.

*itemID2* el índice del segundo ítem a comparar.

*itemData2* el valor del dato del ítem del segundo ítem a comparar.

El valor de retorno debe ser -1, 0 ó 1, dependiendo de si el primer ítem precede al segundo en el orden establecido, si son iguales o si el segundo precede al primero, respectivamente.

Por ejemplo, si para nuestra aplicación establecemos que el orden depende del valor del dato del ítem, de menor a mayor, devolveremos -1 si *itemData1* es menor que *itemData2*, 0 si son iguales y 1 si el valor de *itemData1* es mayor que *itemData2*.

# Capítulo 41 Control Button avanzado

En nuestra primera aproximación a los controles button los dividimos en cuatro categorías: [botones de pulsar \(push buttons\)](#), [botones de grupo](#), [check boxes](#) y [radio buttons](#).

En este capítulo veremos algunos detalles que no vimos en esos capítulos, pero esta vez agrupando todas estas categorías en una sola, ya que todas ellas son en realidad botones de diferentes estilos.

## Insertar botones durante la ejecución

Al igual que vimos con los controles edit y list box, también es posible insertar controles button durante la ejecución. En el caso del control button tendremos que insertar una ventana de la clase "BUTTON". Para insertar el control también usaremos las funciones [CreateWindow](#) y [CreateWindowEx](#).

```
HWND hctrl;  
...  
hctrl = CreateWindowEx(  
    0,  
    "BUTTON",           /* Nombre de la clase */  
    "Botón 1",          /* Texto del título */  
    BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_BORDER |  
    WS_TABSTOP, /* Estilo */  
    9, 19,              /* Posición */  
    95, 24,             /* Tamaño */  
    hwnd,               /* Ventana padre */  
    (HMENU)ID_BOTON,    /* Identificador del control */  
    hInstance,          /* Instancia */  
    NULL);              /* Sin datos de creación de ventana
```



```
*/  
SetFocus(hctrl);
```

A diferencia de los controles que hemos visto antes, en el caso de los botones el texto del título sí tiene uso, ya que será ese texto el que aparezca en el botón, radio button, check box o group box.

El identificador del control se suministra a través del parámetro *hMenu*, por lo que será necesario hacer un casting de ese valor a **HMENU**.

Ahora será nuestro procedimiento de ventana, si el control fue insertado en una ventana, o el procedimiento de diálogo, si se trata de un diálogo, el encargado de procesar los mensajes procedentes del control.

## Cambiar fuente

También es posible modificar la fuente de un control button enviando un mensaje **WM\_SETFONT**. El lugar apropiado es, por supuesto, al procesar el mensaje **WM\_INITDIALOG** al iniciar un cuadro de diálogo, o al procesar el mensaje **WM\_CREATE**, al iniciar una ventana.

En el parámetro *wParam* pasamos un manipulador de fuente, y usaremos la macro **MAKELPARAM** para crear un valor **LPARAM**, en el que especificaremos la opción de repintar el control, que se almacena en la palabra de menor peso de **LPARAM**.

Esto nos permite modificar la fuente durante la ejecución, reflejando los cambios en pantalla.

```
static HFONT hfont;  
...  
hfont = (HFONT)GetStockObject( DEFAULT_GUI_FONT );  
SendMessage(hctrl, WM_SETFONT, (LPARAM)hfont,  
MAKELPARAM(TRUE, 0));
```

## Cambiar colores

Análogamente a lo que hemos visto con otros controles, también existe un mensaje que nos permite modificar el color de los controles de tipo botón.

Se trata del mensaje `WM_CTLCOLORBTN`, que se envía a la ventana propietaria del control cuando debe ser dibujado.

En el parámetro `wParam` recibiremos un manipulador de contexto de dispositivo del control, y en el parámetro `lParam` el manipulador ventana del control. Podemos cambiar el color de fondo y el del texto, y cuando se procese este mensaje, y deberemos retornar un manipulador de pincel, que se usará para pintar el fondo:

```
static HBRUSH pincel;
...
case WM_CREATE:
    pincel = CreateSolidBrush( RGB(0,255,0) );
    ...
case WM_CTLCOLORBTN:
    SetTextColor( (HDC)wParam, RGB(0,0,255) );
    SetBkColor( (HDC)wParam, RGB(0,255,0) );
    return (LRESULT)pincel;
case WM_DESTROY:
    DeleteObject(pincel);
    ...
```

Lo que pasa es que, en la mayor parte de los controles botón, los colores están predefinidos por el sistema, y nuestros cambios al procesar este mensaje no influyen en el aspecto final de los controles.

La excepción son los controles botón del estilo `owner-draw`, sólo en este tipo de botones puede ser útil procesar este mensaje.

## Modificar el bucle de mensajes

Al insertar controles botón en la ventana principal, en lugar de hacerlo en un cuadro de diálogo, hay determinadas funcionalidades, relacionadas con el teclado, que no funcionan de forma similar.

Por ejemplo, para hacer que funcione la tecla de tabulación de modo que cambie el foco entre los distintos controles o para que las teclas como el espacio o el ENTER activen los botones, y para que funcionen las teclas del cursor, hay que modificar el bucle de mensajes de modo que ciertos mensajes, aquellos propios de los cuadros de diálogo, se procesen de forma diferente.

Esto se hace añadiendo la función `IsDialogMessage`. Esta función averigua si un mensaje es de diálogo, y en ese caso lo procesa. Estos mensaje no deben ser procesados por el bucle habitual:

```
/* Bucle de mensajes, se ejecuta hasta que haya error o
GetMessage devuelva FALSE
Modificado para procesar ciertas teclas de forma
automática. */
while(TRUE == GetMessage(&mensaje, NULL, 0, 0)) {
    if(!IsDialogMessage(hwnd, &mensaje) ) {
        /* Traducir mensajes de teclas virtuales a mensajes
de caracteres */
        TranslateMessage(&mensaje);
        /* Enviar mensaje al procedimiento de ventana */
        DispatchMessage(&mensaje);
    }
}
```

## Botones con iconos o mapas de bits

Existen dos estilos para los botones que nos permiten usar un gráfico, un icono o un mapa de bits, en lugar de un texto.

El estilo `BS_ICON` permite asignar un icono a un botón.

El estilo `BS_BITMAP` permite asignar un mapa de bits a un botón.

Estas asignaciones son independientemente de otros estilos que definan el tipo, es decir, se puede asignar un icono o un mapa de bits a un botón pulsable, a un check box o a un radio button.

Además de indicar el estilo al crear el control, o en la definición del recurso, es necesario asignar la imagen al botón. Para eso se usa el mensaje **BM\_SETIMAGE**. En el parámetro wParam se indica el tipo de imagen, **IMAGE\_ICON** para un icono o **IMAGE\_BITMAP** para un mapa de bits. En el mensaje lParam pasaremos el manipulador de la imagen, que será de tipo **HICON** para un icono y de tipo **HBITMAP** para un mapa de bits.

Ejemplo de botón con un icono:

```
HWND hctrl;
HICON hIcono;
...
hctrl = CreateWindowEx(
    0,
    "BUTTON",          /* Nombre de la clase */
    "icono",           /* Texto del título */
    BS_ICON | BS_AUTOCHECKBOX | WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_TABSTOP, /* Estilo */
    9, 49,             /* Posición */
    95, 24,            /* Tamaño */
    hwnd,              /* Ventana padre */
    (HMENU)ID_BOTON2,  /* Identificador del control */
    hInstance,         /* Instancia */
    NULL);             /* Sin datos de creación de ventana
*/
hIcono = LoadIcon(hInstance, "Icono");
SendMessage(hctrl, BM_SETIMAGE, (WPARAM) IMAGE_ICON,
(LPARAM) hIcono);
```

Un ejemplo de botón con un mapa de bits:

```
HWND hctrl;
HBITMAP hBitmap;
...
hctrl = CreateWindowEx(
    0,
```

```

        "BUTTON",          /* Nombre de la clase */
        "Bitmap",          /* Texto del título */
        BS_BITMAP | WS_CHILD | WS_VISIBLE | WS_BORDER |
WS_TABSTOP, /* Estilo */
        9, 9,              /* Posición */
        87+4, 20+4,        /* Tamaño */
        hwnd,              /* Ventana padre */
        (HMENU)ID_BOTON1, /* Identificador del control */
        hInstance,         /* Instancia */
        NULL);             /* Sin datos de creación de ventana
*/
        hBitmap = LoadBitmap(hInstance, "power");
        SendMessage(hwnd, BM_SETIMAGE, (WPARAM) IMAGE_BITMAP,
(LPARAM) hBitmap);

```

De forma simétrica, podemos usar el mensaje [BM\\_GETIMAGE](#) para obtener un manipulador del icono o mapa de bits asociado a un control botón. Indicaremos en el parámetro wParam el tipo de imagen a recuperar:

```

        HICON hIcono;
        ...
        hIcono = (HICON)SendMessage(hwnd, BM_GETIMAGE,
IMAGE_ICON, 0);

```

## Otros estilos para botones

Generalmente, nuestros botones tendrán el estilo [BS\\_TEXT](#), que es el estilo por defecto y el que se usa si no se indican los estilos [BS\\_ICON](#), [BS\\_BITMAP](#) o [BS\\_OWNERDRAW](#).

El estilo [BS\\_TEXT](#) indica que se trata de un botón normal, con un texto que indica (o debería indicar) la acción del botón.

El estilo [BS\\_MULTILINE](#) es una variación de [BS\\_TEXT](#), que permite fragmentar el texto en varias líneas diferentes, que se amoldan al espacio disponible del botón.

## Alineación de contenidos

Hasta ahora sólo hemos mencionado el estilo `BS_CENTER`, que nos permitía situar el texto, icono o mapa de bits de un botón centrado horizontalmente.

Existen otros estilos que nos permiten situar el contenido en otros lugares. Tres de esos estilos permiten definir la alineación del texto en el sentido horizontal: `BS_LEFT` a la izquierda, `BS_CENTER` en el centro y `BS_RIGHT` a la derecha.

Otros tres estilos permiten definir la alineación en sentido vertical: `BS_TOP` en la parte superior, `BS_VCENTER` centrado verticalmente y `BS_BOTTOM` en la parte inferior.

Estos estilos se pueden combinar, eligiendo uno de cada grupo, de modo que podemos elegir nueve posiciones diferentes para situar el texto:

<code>BS_LEFT   BS_TOP</code>	<code>BS_CENTER   BS_TOP</code>	<code>BS_RIGHT   BS_TOP</code>
<code>BS_LEFT   BS_VCENTER</code>	<code>BS_CENTER   BS_VCENTER</code>	<code>BS_RIGHT   BS_VCENTER</code>
<code>BS_LEFT   BS_BOTTOM</code>	<code>BS_CENTER   BS_BOTTOM</code>	<code>BS_RIGHT   BS_BOTTOM</code>

## Check box y Radio buttons

Por último, existen otros dos estilos, que son equivalentes: `BS_LEFTTEXT` y `BS_RIGHTBUTTON`, que permiten situar el gráfico de botones con estilos check box o radio buttons a la derecha, en lugar de situarlo a la izquierda, que es la posición por defecto.

## Ejemplo 68

## Mensajes de notificación

La clase `Button` dispone de muchos mensajes de notificación, pero la mayor parte de ellos se mantienen sólo por compatibilidad

con versiones de Windows de 16 bits, y no deben usarse en aplicaciones de 32 bits, por lo que no los veremos con detalle.

**Nota:**

Los mensajes de notificación obsoletos son: [BN\\_DBLCLK](#), [BN\\_DISABLE](#), [BN\\_DOUBLECLICKED](#), [BN\\_HILITE](#), [BN\\_PAINT](#), [BN\\_PUSHED](#), [BN\\_UNHILITE](#) y [BN\\_UNPUSHED](#). En lugar de usar estos mensajes, las aplicaciones de 32 bits deben usar un control botón con un estilo owner-draw, y la estructura [DRAWITEMSTRUCT](#).

Nos limitaremos a explicar, por lo tanto, sólo tres mensajes de notificación.

Como en todos los casos, los mensajes de notificación se envían a la ventana padre mediante un mensaje [WM\\_COMMAND](#).

## Selección

Cada vez que el usuario hace clic sobre un botón, se envía un mensaje de notificación [BN\\_CLICKED](#) a la ventana propietaria del botón.

## Doble clic

Cuando el usuario hace un doble clic sobre un botón se envía un mensaje de notificación [BN\\_DBLCLK](#). El botón debe tener el estilo [BS\\_OWNERDRAW](#) o [BS\\_RADIOBUTTON](#).

## Pérdida y recuperación de foco

Cuando un control pierde el foco del teclado, se envía un mensaje de notificación [BN\\_KILLFOCUS](#) a la ventana propietaria. Análogamente, cuando un control botón recupera el foco del

teclado, se envía un mensaje `BN_SETFOCUS` a la ventana propietaria.

## Inhibir mensajes de notificación

Si definimos un botón sin el estilo `BS_NOTIFY`, los mensajes `BN_DISABLE`, `BN_PUSHED`, `BN_KILLFOCUS`, `BN_PAINT`, `BN_SETFOCUS` y `BN_UNPUSHED` no se enviarán a la ventana padre del control.

Esto significa que deberemos ser cuidadosos cuando procesemos los mensajes `WM_COMMAND` procedentes de un botón con el estilo `BS_NOTIFY`, ya que no todos los mensajes `WM_COMMAND` que recibamos serán pulsaciones de botón, y por lo tanto no nos servirá el método usado en el [capítulo 9](#), en el que no se verificaba el valor del parámetro `wParam`.

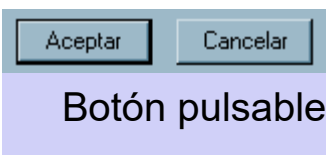
Por contra, un botón pulsable sin el estilo `BS_NOTIFY` sólo puede enviar mensajes de notificación `BN_CLICKED`, de modo que no tiene sentido verificar el valor de la palabra de mayor peso del parámetro `wParam`.

Tanto si se usa el estilo `BS_NOTIFY`, como si no, los mensajes `BN_CLICKED` y `BN_DBLCLK` siempre se envían.

## Estilos de cada tipo de botón

Ya hemos mencionado que los botones pulsables, los check boxes, los radio buttons y las cajas de grupos no son más que controles botón con diferentes estilos. Veamos ahora qué estilos puede tener cada uno de los controles:

### Botones pulsables



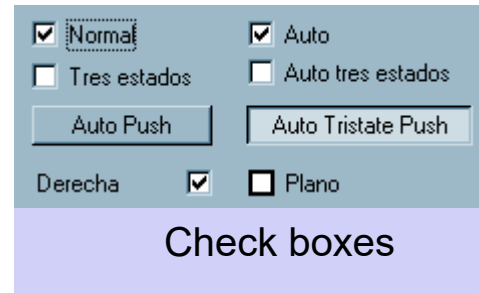
Estos botones se pueden definir usando los estilos `BS_PUSHBUTTON` y `BS_DEFPUSHBUTTON`. En el segundo caso,



el botón será el botón por defecto, y se activará cuando el usuario pulse la tecla de ENTER.

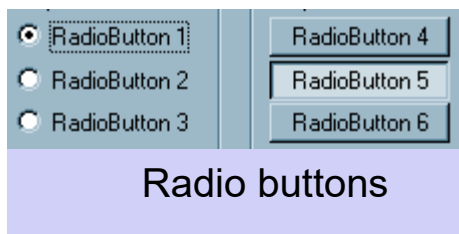
## Check boxes

Los estilos [BS\\_CHECKBOX](#), [BS\\_AUTOCHECKBOX](#), [BS\\_3STATE](#) y [BS\\_AUTO3STATE](#) definen botones check box de dos estados (los dos primeros) o de tres estados (los dos últimos). Los estilos AUTO, además, se procesarán automáticamente por el sistema.



Añadir que el estilo [BS\\_PUSHLIKE](#) permite definir un check box con la misma apariencia que un botón pulsable.

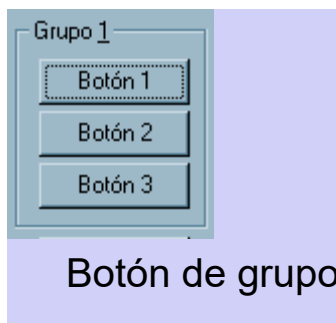
## Radio buttons



Para definir este tipo de botones disponemos de los estilos [BS\\_RADIOBUTTON](#) y [BS\\_AUTORADIOBUTTON](#). El segundo define radio buttons automáticos.

En este caso, el estilo [BS\\_PUSHLIKE](#) también permite definir un radio buttons con la misma apariencia que un botón pulsable.

## Cajas de grupo



El estilo [BS\\_GROUPBOX](#) permite definir cajas de grupo.

Todos estos controles, menos los group boxes, envían mensajes [WM\\_COMMAND](#) cuando son pulsados, y mensajes de notificación [BN\\_CLICKED](#).

## Botones owner-draw

Existe un quinto tipo de botón, el owner-draw, que se define con el estilo `BS_OWNERDRAW`. El comportamiento de estos botones depende de nosotros, ya que estaremos obligados a diseñarlos y pintarlos dependiendo de su estado.

## Estados de un botón

Un botón puede tener tres niveles de estado diferentes.

Por una parte, puede tener o no el foco del teclado. Esto se indica, generalmente, mediante un rectángulo de líneas punteadas alrededor del texto.

Puede estar o no resaltado. El resaltado se produce cuando el usuario coloca el ratón sobre el botón y pulsa, y mantiene, el botón izquierdo del ratón. O también si el control tiene el foco y se pulsa la tecla [Espacio].

Por último, en el caso de check boxes y radio buttons, el botón puede o no estar marcado (checked).

## Selección de un botón

En los botones que usaremos normalmente, tanto el estado del control como su aspecto gráfico se actualizarán de forma automática por el sistema. Esto se aplica a botones pulsables, que cambiarán de aspecto cuando se pulsen y se suelten y también a los check boxes y radio buttons automáticos, que mostrarán el estado de marcado de forma automática.

En los radio buttons y check boxes no automáticos, el estado de marcado dependerá de nuestra aplicación. Esto nos permite crear condiciones más elaboradas para el cambio de estado que las simples pulsaciones.

Por último, en los botones owner draw, tanto el cambio de estado como la actualización de la representación dependerán de nuestra

aplicación. Esto nos dará mucho más trabajo, pero a cambio proporciona una total libertad en cuanto al comportamiento y aspecto de los botones.

## Cambios de estado

Para determinar el estado de un botón se usan los mensajes [BM\\_GETCHECK](#) y [BM\\_GETSTATE](#).

El mensaje [BM\\_GETCHECK](#) puede devolver los valores [BST\\_CHECKED](#), [BST\\_INDETERMINATE](#) o [BST\\_UNCHECKED](#), de modo que podemos determinar el estado de marcado de un botón check box o radio button. El valor [BST\\_INDETERMINATE](#) sólo es válido para botones con el estilo [BS\\_3STATE](#) o [BS\\_AUTO3STATE](#).

El mensaje [BM\\_GETSTATE](#) es algo más completo, ya que puede devolver los valores [BST\\_FOCUS](#) y [BST\\_PUSHED](#), además de los tres que devuelve [BM\\_GETCHECK](#).

Podemos usar la máscara 0x0003, o mejor aún, la suma de los valores [BST\\_CHECKED](#) y [BST\\_INDETERMINATE](#), para extraer sólo los valores de la marca de chequeo. Aunque esto es redundante, ya que podemos usar en su lugar el mensaje [BM\\_GETCHECK](#).

El valor [BST\\_FOCUS](#) indica si el botón tiene el foco del teclado, y el valor [BST\\_PUSHED](#) si el botón está pulsado.

```
if(SendDlgItemMessage(hwnd, ID_BOTON2, BM_GETSTATE, 0, 0)
== BST_CHECKED)
    MessageBox(hwnd, "Botón 2 marcado", "Marca", MB_OK);
else
    MessageBox(hwnd, "Botón 2 no marcado", "Marca",
MB_OK);
```

De forma simétrica, podemos usar los mensajes [BM\\_SETCHECK](#) o [BM\\_SETSTATE](#) para modificar el estado de un botón.

[BM\\_SETCHECK](#) nos permite modificar el estado de marcado de un botón del tipo Check Box o Radio Button, usando uno de los

valores **BST\_CHECKED**, **BST\_INDETERMINATE** o **BST\_UNCHECKED**, en el parámetro wParam.

El mensaje **BM\_SETSTATE** permite modificar el estado del resaltado de un botón. Para ello se usa el parámetro wParam. Un valor **TRUE** resalta el botón, y un valor **FALSE** elimina el resalte. Estos estados no afectan más que a la apariencia del botón. El estado resaltado corresponde a cuando el usuario pulsa y mantiene pulsado un botón.

```
SendDlgItemMessage(hwnd, ID_BOTON2, BM_SETSTATE, TRUE, 0);  
SendDlgItemMessage(hwnd, ID_BOTON2, BM_SETCHECK, BST_UNCHECKED, 0);
```

## Funciones para controles botón

Como en el resto de los controles, los mensajes se pueden enviar mediante dos funciones distintas.

Por una parte, la función **SendMessage** nos permite enviar un mensaje a una ventana o control, disponiendo de su manipulador de ventana:

```
SendMessage(hwnd, WM_SETFONT, (LPARAM)hfont, MAKELPARAM(TRUE, 0));
```

Si no disponemos de tal manipulador existen otras opciones. Por ejemplo, podemos obtener un manipulador de ventana o control, a partir del identificador, mediante la función **GetDlgItem**. Combinando estas dos funciones, podemos enviar un mensaje a un control, aunque no dispongamos de un manipulador de ventana:

```
SendMessage(GetDlgItem(hwnd, ID_BOTON1), WM_SETFONT,
```

```
(WPARAM)hfont, MAKELPARAM(TRUE, 0));
```

Sin embargo, existe una función que combina las acciones de estas dos, se trata de [SendDlgItemMessage](#):

```
SendDlgItemMessage(hwnd, ID_BOTON1, WM_SETFONT,  
(WPARAM)hfont, MAKELPARAM(TRUE, 0));
```

A pesar de que aparezca la abreviatura "Dlg" como parte de estas dos últimas funciones, ambas funcionan tanto en cuadros de diálogo como en ventanas normales. Es decir, podemos usar estas dos funciones para enviar mensajes a controles insertados en ventanas corrientes.

## Funciones propias de controles botón

Otras funciones, que ya vimos previamente en los capítulos [14](#) y [15](#), son específicas para controles botón de los tipos check box y radio button.

Aunque estas funciones tienen sus equivalentes en mensajes, a menudo nos serán útiles en nuestros programas.

La función [CheckDlgButton](#), nos permite cambiar el estado de marcado de un control botón del tipo check box o radio button, en realidad equivale a enviar un mensaje [BM\\_SETCHECK](#).

La función [CheckRadioButton](#) es más útil, ya que trabaja con grupos de radio buttons. Dentro de un grupo de radio buttons sólo uno de ellos puede estar marcado en un momento dado. Usar esta función nos permite marcar uno de los controles, y eliminar la marca del que la tenía previamente, en una única operación.

La función [IsDlgButtonChecked](#) nos permite conocer el estado de marcado de un control botón. Es equivalente a enviar un mensaje [BM\\_GETCHECK](#) al control.

## Modificar el estilo de un botón

Es posible modificar el estilo de un botón durante la ejecución. Para ello se usa el mensaje `BM_SETSTYLE`, indicando en el parámetro `wParam` el nuevo estilo del botón y en `lParam` si se debe o no redibujar el control, un valor `TRUE` en la palabra de menor peso indica que se debe redibujar, un valor `FALSE`, que no:

```
SendDlgItemMessage(hwnd, ID_BOTON, BM_SETSTYLE,  
    BS_PUSHBUTTON | BS_LEFTTEXT | WS_CHILD | WS_VISIBLE |  
    WS_TABSTOP,  
    MAKELPARAM(TRUE, 0));
```

## Botones owner-draw

Ya lo hemos mencionado antes: los controles botón también disponen de un estilo owner-draw.

Un control botón con el estilo owner-draw, `BS_OWNERDRAW` no tiene un comportamiento definido, ni un aspecto gráfico concreto. Será el procedimiento de ventana o diálogo de la ventana propietaria del control el encargado de actualizar el aspecto en pantalla y también de definir las respuestas a cada evento de teclado o ratón.

De modo que un botón owner-draw puede ser un botón pulsable, un radio button, un check box o cualquier otra cosa que inventemos.

Lo primero es crear el control con el estilo `BS_OWNERDRAW`:

```
HWND hctrl;  
...  
hctrl = CreateWindowEx(  
    0,  
    "BUTTON",          /* Nombre de la clase */  
    "Botón 2",         /* Texto del título */  
    BS_OWNERDRAW | WS_CHILD | WS_VISIBLE | WS_BORDER |
```

```

WS_TABSTOP, /* Estilo */
    9, 49,      /* Posición */
    95, 24,     /* Tamaño */
    hwnd,       /* Ventana padre */
    (HMENU)ID_BOTON, /* Identificador del control */
    hInstance,  /* Instancia */
    NULL);      /* Sin datos de creación de ventana
*/

```

Cuando existen botones con el estilo owner-draw, la ventana padre del control recibirá un mensaje **WM\_DRAWITEM** cada vez que el control deba ser redibujado. Este mensaje se recibe para todos los controles owner-draw existentes, de modo que deberemos distinguir a qué control en concreto se refiere el mensaje. Esta es la parte sencilla, ya que en el parámetro *wParam* de este mensaje recibiremos el identificador del control.

Por otra parte, en el parámetro *lParam*, recibiremos un puntero a una estructura **DRAWITEMSTRUCT**, que nos proporciona información sobre todos los detalles necesarios para decidir el modo en que debemos dibujar el control.

Por una parte, el campo *CtlType* contendrá el valor **ODT\_BUTTON**, indicando que el control es un botón. El campo *CtlID* contiene el identificador del control. El campo *itemID* no tiene ninguna información en el caso de un control botón. El campo *itemAction* contiene un valor que especifica el tipo de acción de dibujo requerido. Puede ser **ODA\_DRAWENTIRE**, si se necesita actualizar el control completo, **ODA\_FOCUS** si el único cambio es la pérdida o recuperación del foco o **ODA\_SELECT**, si ha cambiado el estado de selección.

El campo *itemState* especifica el estado del control. En el caso de los botones, el valor de este campo puede ser una combinación de: **ODS\_DISABLED**, si el control está deshabilitado, **ODS\_FOCUS**, si el control tiene el foco o **ODS\_SELECTED**, si el control está seleccionado.

El campo *hwndItem* contiene el manipulador de ventana, *hDC* contiene el manipulador de contexto de dispositivo del control. El

campo *rcItem* contiene el rectángulo que define los límites de la zona a dibujar. E *itemData* no contiene nada válido en el caso de los controles botón.

Este ejemplo visualiza un control botón con forma elíptica, y cambia el color del fondo a naranja cuando está pulsado. Si se deshabilita, se muestra una cruz:

```
LPDRAWITEMSTRUCT lpdis;
HBRUSH pincel, pincel2;
...
case WM_DRAWITEM:
    lpdis = (LPDRAWITEMSTRUCT)lParam;
    GetClientRect(lpdis->hwndItem, &re);
    SetBkMode(lpdis->hDC, TRANSPARENT);
    pincel =
CreateSolidBrush(GetSysColor(COLOR_BACKGROUND));
    pincel2 = CreateSolidBrush(RGB(240,120,0));
    FillRect(lpdis->hDC, &re, pincel);
    if(wParam == ID_BOTON2) {
        if(lpdis->itemState & ODS_SELECTED) {
            SelectObject(lpdis->hDC,
(HBRUSH)GetStockObject(GRAY_BRUSH));
            SelectObject(lpdis->hDC,
(HPEN)GetStockObject(WHITE_PEN));
            Ellipse(lpdis->hDC, re.left, re.top, re.right,
re.bottom);
            TextOut(lpdis->hDC, re.left+14, re.top+4, "Botón
2", 7);
        } else if(lpdis->itemState & ODS_DISABLED) {
            MoveToEx(lpdis->hDC, 0, 0, NULL);
            LineTo(lpdis->hDC, re.right, re.bottom);
            MoveToEx(lpdis->hDC, 0, re.bottom, NULL);
            LineTo(lpdis->hDC, re.right, 0);
        } else {
            SelectObject(lpdis->hDC, pincel2);
            SelectObject(lpdis->hDC,
(HPEN)GetStockObject(WHITE_PEN));
            Ellipse(lpdis->hDC, re.left, re.top, re.right,
re.bottom);
            TextOut(lpdis->hDC, re.left+12, re.top+2, "Botón
2", 7);
        }
    }
}
```



```
DeleteObject(pincel);  
DeleteObject(pincel2);  
return 0;
```

## Ejemplo 69

# Capítulo 42 Control estático avanzado

En el [capítulo 10](#) ya vimos la mayor parte de lo que se puede decir sobre los controles estáticos. En este capítulo veremos que estos controles no son tan poco interactivos como parecen, y aunque no pueden ser seleccionados, ni pueden recibir el foco, ni responden al teclado; sí puede recibir algunos mensajes de notificación, si se definen con el estilo adecuado.

## Insertar controles estáticos durante la ejecución

Empezaremos por ver cómo insertar controles estáticos durante la ejecución. Esto ya no es nada nuevo, el proceso es similar al de los controles que ya hemos visto, la única diferencia es que en este caso se trata de ventanas de la clase `STATIC`. Para insertar el control también usaremos las funciones [CreateWindow](#) y [CreateWindowEx](#).

```
HWND hctrl;  
...  
hctrl = CreateWindowEx(  
    0,  
    "STATIC",           /* Nombre de la clase */  
    "Texto",            /* Texto del título */  
    SS_SIMPLE |  
    WS_CHILD | WS_VISIBLE, /* Estilo */  
    5, 5,               /* Posición */  
    55, 55,             /* Tamaño */  
    hwnd,               /* Ventana padre */  
    (HMENU)ID_ESTAT,    /* Identificador del control */
```

```
hInstance,          /* Instancia */  
NULL);              /* Sin datos de creación de ventana  
*/
```

En aquellos controles estáticos que visuelicen texto, el parámetro del título se usará para indicar el texto. En otros casos podrá ser un nombre de fichero, y en el resto se ignora.

El identificador del control se suministra a través del parámetro *hMenu*, por lo que será necesario hacer un casting.

Sin embargo, con frecuencia no será necesario procesar mensajes de estos controles, ni enviárselos durante la ejecución, de modo que muchas veces podemos usar el mismo identificador para todos ellos. En esos casos no existe ningún valor en especial para usar como identificador, aunque suele usarse el valor -1.

## Cambiar fuente

También es posible modificar la fuente de un control estático enviando un mensaje [WM\\_SETFONT](#). El lugar apropiado es, por supuesto, al procesar el mensaje [WM\\_INITDIALOG](#) en diálogos o al iniciar la ventana, al procesar el mensaje [WM\\_CREATE](#).

En el parámetro *wParam* pasamos un manipulador de fuente, y usaremos la macro [MAKELPARAM](#) para crear un valor [LPARAM](#), en el que especificaremos la opción de repintar el control, que se almacena en la palabra de menor peso de [LPARAM](#).

Esto nos permite modificar la fuente durante la ejecución, reflejando los cambios en pantalla.

```
static HFONT hfont;  
...  
hfont = (HFONT)GetStockObject( DEFAULT_GUI_FONT );  
SendMessage(hctrl, WM_SETFONT, (LPARAM)hfont,  
MAKELPARAM(TRUE, 0));
```

## Cambiar colores

Análogamente a lo que hemos visto con otros controles, también existe un mensaje que nos permite modificar el color de los controles de tipo estático.

Se trata del mensaje [WM\\_CTLCOLORSTATIC](#), que se envía a la ventana propietaria del control cuando debe ser dibujado.

En el parámetro wParam recibiremos un manipulador de contexto de dispositivo del control, y en el parámetro lParam el manipulador ventana del control. Podemos cambiar el color de fondo y el del texto, y cuando se procese este mensaje, y deberemos retornar un manipulador de pincel, que se usará para pintar el fondo:

```
static HBRUSH pincel;
...
case WM_CREATE:
    pincel = CreateSolidBrush( RGB(0,255,0) );
    ...
case WM_CTLCOLORSTATIC:
    SetTextColor( (HDC)wParam, RGB(0,0,255) );
    SetBkColor( (HDC)wParam, RGB(0,255,0) );
    return (LRESULT)pincel;
case WM_DESTROY:
    DeleteObject(pincel);
    ...
```

## Estilos estáticos gráficos

Recordemos ahora los diferentes estilos de controles estáticos, y aprovechemos para ver algunos que no vimos anteriormente:

### Marcos

- [SS\\_BLACKFRAME](#): marco de color negro.
- [SS\\_GRAYFRAME](#): marco de color gris.
- [SS\\_WHITEFRAME](#): marco de color blanco.

## Rectángulos

- [SS\\_BLACKRECT](#): rectángulo negro.
- [SS\\_GRAYRECT](#): rectángulo gris.
- [SS\\_WHITERECT](#): rectángulo blanco.

## Ranurados

- [SS\\_ETCHEDHORZ](#): ranura horizontal.
- [SS\\_ETCHEDVERT](#): ranura vertical.
- [SS\\_ETCHEDFRAME](#): rectángulo ranurado.

## Ejemplos

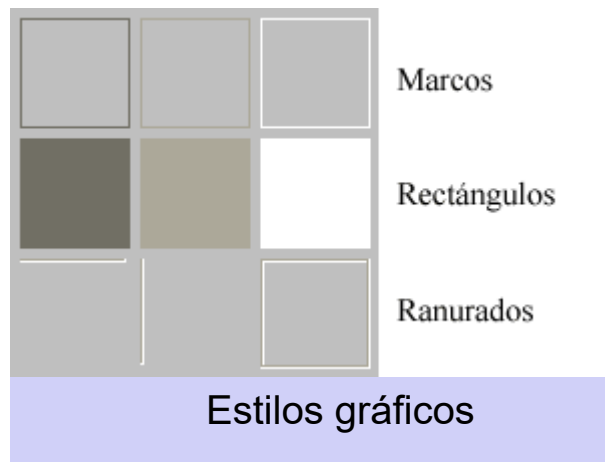
### Más sobre los ranurados

Los ranurados se consiguen mediante el uso de la función [DrawEdge](#), que también está disponible para usarse en la decoración de ventanas y cuadros de diálogo.

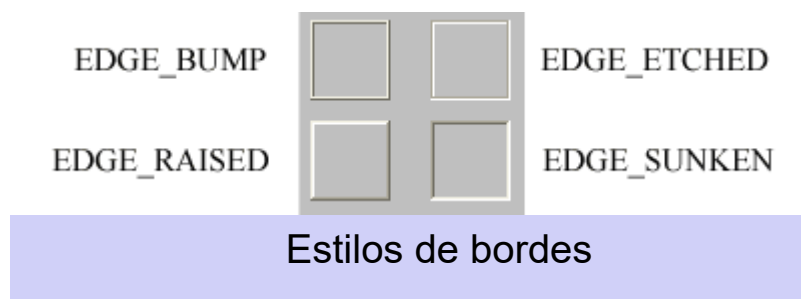
No se trata de una función del GDI, sino de una función de *user32*, por lo que este es, probablemente, el mejor sitio para comentar su uso.

Esta función tiene cuatro parámetros:

- El primero es un manipulador del DC, donde se trazará el ranurado.
- El segundo es un puntero a una estructura [RECT](#) con las coordenadas lógicas del rectángulo a ranurar.
- El tercero sirve para especificar qué tipo de borde queremos trazar para el interior y exterior. Hay dos opciones para el borde interior: [BDR\\_RAISEDINNER](#) y [BDR\\_SUNKENINNER](#) (hacia afuera y hacia adentro, respectivamente); y dos opciones para



el exterior: **BDR\_RAISEDOUTER** y **BDR\_SUNKENOUTER**. Se debe cambiar una opción interior con una exterior, de modo que, hay cuatro posibilidades, para las que también existen macros definidas: **EDGE\_BUMP**, **EDGE\_ETCHED**, **EDGE\_RAISED** y **EDGE\_SUNKEN**, (reborde, ranurado, elevado y hundido).



- El cuarto parámetro sirve para indicar el tipo de borde, y también pueden ser combinados para obtener más tipos. Para ver cada tipo

individual se puede usar el programa de ejemplo.

Algunos de los tipos se refieren a los lados del rectángulo que se van a dibujar, por ejemplo, **BF\_RIGHT**, **BF\_LEFT**, **BF\_TOP** y **BF\_BOTTOM**, o combinaciones de ellos, como **BF\_BOTTOMLEFT**, **BF\_BOTTOMRIGHT**, **BF\_TOPLEFT** y **BF\_TOPRIGHT**. Estos estilos se pueden combinar entre si para obtener las combinaciones que faltan, usando el operador de bits **|**. También se puede usar **BF\_RECT** para mostrar los cuatro lados del rectángulo.

Otros tipos permiten trazar diagonales: **BF\_DIAGONAL**, **BF\_ENDTOPRIGHT**, etc.

El resto permite definir diferentes aspectos, por ejemplo, **BF\_FLAT** crea ranurados planos, que no dan sensación de relieve; **BF\_MIDDLE** muestra la superficie interior del rectángulo a un nivel intermedio de profundidad, usando un tono de gris intermedio; **MF\_MONO** elimina por completo el relieve 3D; **BF\_SOFT** da un acabado blando, redondeado; **BF\_ADJUST** crea el ranurado de modo que se deje libre el área interna del rectángulo para que la use el cliente.

## Estilos estáticos de texto

Entre los estilos para controles estáticos de texto tenemos los siguientes:

- **SS\_SIMPLE**: la forma simple alinea el texto a la izquierda, y descartará la parte que no quepa en el rectángulo especificado.
- **SS\_LEFTNOWORDWRAP**: igual que el estilo **SS\_SIMPLE**, salvo que los caracteres de tabulación se expanden.
- **SS\_LEFT**: igual que el estilo **SS\_LEFTNOWORDWRAP**, salvo que el texto se dividirá en tantas líneas como sea necesario para visualizarlo por completo.
- **SS\_CENTER**: igual que **SS\_LEFT**, salvo que cada línea de texto se centrará horizontalmente en el rectángulo dado.
- **SS\_RIGHT**: igual que **SS\_LEFT**, salvo que las líneas se alinean a la derecha.

Ejemplo de texto para control estático	Estilo SS_SIMPLE	Se puede modificar el texto dentro de un control estático mediante la función <b>SetWindowText</b> o mediante el mensaje <b>WM_SETTEXT</b> . Recordemos que podemos enviar mensajes a un control
Ejemplo de texto para control estático	Estilo SS_LEFT	
Ejemplo de texto para control estático	Estilo SS_CENTER	
Ejemplo de texto para control estático	Estilo SS_RIGHT	
Ejemplo de texto para control estático	Estilo SS_LEFTNOWORDWRAP	
Estilos de texto		

mediante las funciones **SendDlgItemMessage**, conociendo el identificador del control, o usando **SendMessage**, si disponemos del manipulador de ventana del control. Este manipulador se puede conseguir también mediante la función **GetDlgItem**, conociendo el identificador del control.

Estas tres líneas de código son, por lo tanto, equivalentes:

```
SetWindowText(GetDlgItem(hwnd, ID_EST), "Nuevo texto");
SendDlgItemMessage(hwnd, ID_EST, WM_SETTEXT, 0,
(LPARAM) "Nuevo texto");
SendMessage(GetDlgItem(hwnd, ID_EST), WM_SETTEXT, 0,
(LPARAM) "Nuevo texto");
```

El modificador de estilo **SS\_NOPREFIX** se puede aplicar a cualquiera de los estilos anteriores. Cuando se hace, el resultado es que los caracteres & no se interpretan como prefijos de aceleradores. Por defecto, el carácter & no se muestra, sino que provoca que el siguiente carácter se subraye, y la tecla correspondiente a ese carácter funcione como un acelerador. Cuando este acelerador se usa, el foco pasa al siguiente control que admita el foco del teclado.

## Imágenes

Existen tres estilos básicos para crear controles estáticos con imágenes:

&Ejemplo	Con estilo SS_NOPREFIX
Ejemplo	Sin estilo SS_NOPREFIX
Modificador SS_NOPREFIX	

- **SS\_BITMAP**: se mostrará un mapa de bits. Si se hace desde un fichero de recursos, el texto del control corresponde con un recurso de mapa de bits definido en algún lugar del fichero de recursos. Si se inserta directamente en la ventana o diálogo, será necesario asignar un mapa de bits.
- **SS\_ICON**: se mostrará un icono. Si se hace desde un fichero de recursos, el texto del control corresponde con un recurso de icono definido en algún lugar del fichero de recursos. Si se inserta directamente en la ventana o diálogo, será necesario asignar un icono.
- **SS\_ENHMETAFILE**: se mostrará un metaarchivo mejorado. Si se hace desde un fichero de recursos, el texto del control corresponde con un recurso de metaarchivo definido en algún lugar del fichero de recursos. Si se inserta directamente en la ventana o diálogo, será necesario asignar un metaarchivo.



Noveremos este estilo por ahora, en el futuro estudiaremos los meta ficheros y los meta ficheros mejorados.

## Mensajes para asignar imágenes

Podemos usar el mensaje [STM\\_SETIMAGE](#) para asignar un mapa de bits, un icono, un cursor o un meta fichero a un control estático. El parámetro wParam contendrá una constante que indica el tipo de imagen, y el parámetro lParam un manipulador de imagen:

```
SendMessage(hwnd, STM_SETIMAGE, IMAGE_BITMAP,  
(LPARAM) bitmap);  
SendMessage(hwnd, STM_SETIMAGE, IMAGE_ICON,  
(LPARAM) Icono);
```

De forma simétrica, el mensaje [STM\\_GETIMAGE](#) permite obtener un manipulador de la imagen asociada a un control estático. Igual que antes, el parámetro wParam contendrá una constante que indica el tipo de imagen.

Además, existen otros mensajes equivalentes para controles estáticos con el estilo [SS\\_ICON](#). Así, el mensaje [STM\\_SETICON](#) permite asignar un icono a un control estático, y el mensaje [STM\\_GETICON](#) recuperar un manipulador del icono actual del control. Estas dos líneas son equivalentes:

```
SendMessage(hwnd, STM_SETICON, (WPARAM) Icono, 0);  
SendMessage(hwnd, STM_SETIMAGE, IMAGE_ICON,  
(LPARAM) Icono);
```

## Modificadores de estilo

Podemos aplicar algunos modificadores de estilo a los tres descritos anteriormente. Si no se aplica ningún modificador, el

tamaño del control se calcula a partir del tamaño de la imagen, y se ignora el especificado al crear el control.

- **SS\_CENTERIMAGE**: si se especifica, el control tendrá el tamaño especificado, el gráfico se mostrará centrado y la parte no usada se rellena con el color del pixel de la esquina superior izquierda de la imagen.
- **SS\_REALSIZEIMAGE**: cuando se usa, el gráfico se escalará si el tamaño del control aumenta o disminuye, de modo que ocupe toda su área de cliente.
- **SS\_RIGHTJUST**: hasta la fecha ignoro para qué sirve esta opción. :-)

## Modificador de hundido

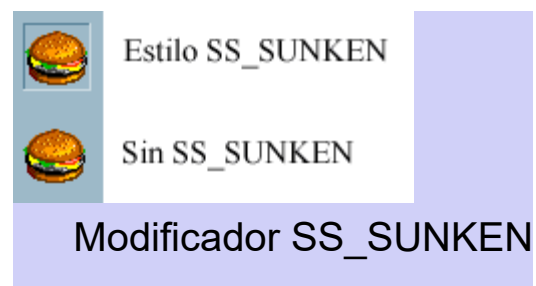
Todos los estilos anteriores se pueden combinar con un modificador, el estilo **SS\_SUNKEN**, que crea la apariencia de que el control está ligeramente hundido con respecto al resto del área de cliente de la ventana:

## Mensajes de notificación

La clase Static dispone de varios mensajes de notificación, pero para que se envíen se debe activar el estilo **SS\_NOTIFY**. En realidad los controles static no están diseñados para ser interactivos, de modo que esto es algo lógico.

Como en todos los casos, los mensajes de notificación se envían a la ventana padre en la palabra de mayor peso del parámetro wParam, mediante un mensaje **WM\_COMMAND**. Los mensajes de notificación son:

- **STN\_CLICKED**: se envía cuando el usuario hace clic sobre el control.



- **STN\_DBLCLK**: se envía cuando el usuario hace sobre clic sobre el control.
- **STN\_DISABLE**: se envía cuando el control es deshabilitado.
- **STN\_ENABLE**: se envía cuando el control es habilitado.

Por ejemplo:

```

        case WM_COMMAND:
            switch(LOWORD(wParam)) {
                case ID_EST14:
                    switch(HIWORD(wParam)) {
                        case STN_CLICKED:
                            MessageBox(hwnd, "Clic", "Controles
estáticos", MB_OK);
                            break;
                        case STN_DBLCLK:
                            MessageBox(hwnd, "Doble clic",
"Controles estáticos", MB_OK);
                            break;
                        case STN_DISABLE:
                            MessageBox(hwnd, "Disable", "Controles
estáticos", MB_OK);
                            break;
                        case STN_ENABLE:
                            MessageBox(hwnd, "Enable", "Controles
estáticos", MB_OK);
                            break;
                    }
                break;
            }
        break;

```

## Controles estáticos owner-draw

Los controles estáticos también disponen de un estilo owner-draw.

Para crear un control estático owner-draw se debe activar el estilo **SS\_OWNERDRAW**. De este modo nuestra aplicación se hará

cargo de mostrar el control, y seremos los únicos responsables de su aspecto gráfico.

Lo primero es crear el control con el estilo `SS_OWNERDRAW`:

```
HWND hctrl;  
...  
hctrl = CreateWindowEx(  
    0,  
    "STATIC",          /* Nombre de la clase */  
    "Texto",           /* Texto del título */  
    SS_OWNERDRAW |  
    WS_CHILD | WS_VISIBLE, /* Estilo */  
    9, 49,             /* Posición */  
    95, 24,            /* Tamaño */  
    hwnd,              /* Ventana padre */  
    (HMENU)ID_STATIC, /* Identificador del control */  
    hInstance,         /* Instancia */  
    NULL);             /* Sin datos de creación de ventana  
*/
```

Cuando existen controles con el estilo owner-draw, la ventana padre del control recibirá un mensaje `WM_DRAWITEM` cada vez que el control deba ser redibujado. Este mensaje se recibe para todos los controles owner-draw existentes, de modo que deberemos distinguir a qué control en concreto se refiere el mensaje. Esta es la parte sencilla, ya que en el parámetro `wParam` de este mensaje recibiremos el identificador del control.

Por otra parte, en el parámetro `lParam`, recibiremos un puntero a una estructura `DRAWITEMSTRUCT`, que nos proporciona información sobre todos los detalles necesarios para decidir el modo en que debemos dibujar el control.

Por una parte, el campo *CtlType* contendrá el valor `ODT_STATIC`, indicando que el control es un control estático. El campo *CtlID* contiene el identificador del control. El campo *itemID* no tiene ninguna información en el caso de un control estático. El campo *itemAction* contiene un valor que especifica el tipo de acción de dibujo requerido. Sólo puede ser `ODA_DRAWENTIRE`, si se

necesita actualizar el control completo, ya que estos controles no pueden tener el foco ni ser seleccionados.

El campo *itemState* especifica el estado del control. En el caso de los controles estáticos, el valor de este campo sólo puede ser: **ODS\_DISABLED**, si el control está deshabilitado.

El campo *hwndItem* contiene el manipulador de ventana, *hDC* contiene el manipulador de contexto de dispositivo del control. El campo *rcItem* contiene el rectángulo que define los límites de la zona a dibujar. E *itemData* no contiene nada válido en el caso de los controles estáticos.

Este ejemplo visualiza un control estático con forma elíptica:

```
LPDRAWITEMSTRUCT lpdis;
case WM_DRAWITEM:
    lpdis = (LPDRAWITEMSTRUCT)lParam;
    SendDlgItemMessage(hwnd, (UINT)wParam,
WM_GETTEXT, 128, (LPARAM)cad);
    GetClientRect(lpdis->hwndItem, &re);
    SetBkMode(lpdis->hDC, TRANSPARENT);
    FillRect(lpdis->hDC, &re,
(HBRUSH)GetStockObject(LTGRAY_BRUSH));
    SelectObject(lpdis->hDC,
(HBRUSH)GetStockObject(GRAY_BRUSH));
    SelectObject(lpdis->hDC,
(HPEN)GetStockObject(WHITE_PEN));
    Ellipse(lpdis->hDC, re.left, re.top, re.right,
re.bottom);
    TextOut(lpdis->hDC, re.left+12, re.top+2, cad,
strlen(cad));
    return 0;
```

## Ejemplo 70

# Capítulo 43 Control combo box avanzado

Seguimos con el repaso de los controles más comunes del API de Window.

En el [capítulo 11](#) vimos cómo usar de una forma básica los combo boxes. También vimos que estos controles son la combinación de un control edit o un control de texto estático y un list box, así como el modo de crearlos a partir de un fichero de recursos, de inicializarlos y de leer el valor de su selección.

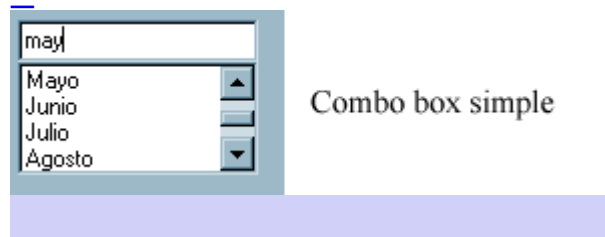
En este capítulo veremos mucho más sobre ellos, cómo personalizarlos, qué estilos pueden tener, sus mensajes de notificación, etc.

Ya que son la combinación de un control edit (o static) y de un control listbox, estos controles tendrán muchas de las características de los controles de los que se derivan, y por lo tanto, disponen de los equivalentes de algunos de sus estilos, mensajes de notificación y mensajes de control.

## Tipos de combo boxes

Ya vimos que existen tres tipos básicos de combo boxes, dependiendo de que se seleccione uno de los tres estilos que los definen:

- Simple: si se usa el estilo [CBS\\_SIMPLE](#)
- Desplegable: (Drop-down) si se usa el estilo [CBS\\_DROPDOWN](#). La lista se puede desplegar mediante el teclado

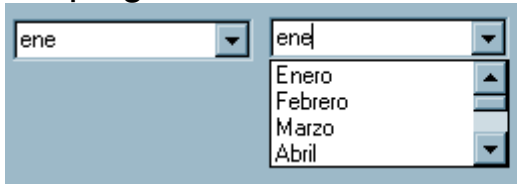


Combo box simple

mediante F4, o pulsando sobre el icono de la flecha

### ComboBox simple

abajo. El texto de entrada se puede introducir en la zona de edición, o se puede elegir un valor de la lista. La siguiente imagen muestra el mismo combo box con la lista plegada y desplegada.



Combo box desplegable

### ComboBox desplegable

- Lista desplegable: (Drop-down list) si se usa

el estilo [CBS\\_DROPDOWNLIST](#). Al igual que en el tipo desplegable, la lista se puede desplegar mediante el teclado o el ratón, pero en este caso, el texto sólo se puede elegir entre los valores de la lista. Se puede usar el teclado para elegir el elemento de la lista, tecleando el principio de la cadena. La imagen siguiente muestra uno de estos controles con la lista plegada y desplegada.



Combo box de lista desplegable

### ComboBox lista desplegable

## Insertar control es combo box durante la ejecución

No hay nada nuevo en este tema, del mismo modo que hemos visto para los controles anteriormente tratados, también es posible insertar controles combo box durante la ejecución. En este caso tendremos que insertar una ventana de la clase "COMBOBOX". Para insertar el control también usaremos las funciones [CreateWindow](#) y [CreateWindowEx](#).

```

        HWND hwnd;
        ...
        case WM_CREATE:
            hwnd = (LPCREATESTRUCT)lParam->hwnd;
            /* Insertar control Edit */
            hwnd = CreateWindowEx(
                0,
                "COMBOBOX",          /* Nombre de la clase */
                NULL,                 /* Texto del título */
                CBS_SIMPLE |
                WS_VSCROLL | WS_CHILD | WS_VISIBLE |
                WS_TABSTOP, /* Estilo */
                5, 5,                 /* Posición */
                120, 85               /* Tamaño */
                hwnd,                 /* Ventana padre */
                (HMENU)ID_COMBO1, /* Identificador del control
            */
                hwnd,                 /* Instancia */
                NULL);               /* Sin datos de creación de
        ventana */

```

Como vemos, usamos los mismos valores que en el fichero de recursos: un identificador, la clase de ventana (en este caso "COMBOBOX"), una combinación de estilos, la posición y las dimensiones.

Al igual que en los demás controles, el identificador del control se suministra a través del parámetro *hMenu*, por lo que será necesario hacer un casting a [HMENU](#).

## Cambiar la fuente de un control combo box

En esto tampoco hay novedades, podemos cambiar la fuente de un control combo box mediante un mensaje [WM\\_SETFONT](#):

```

        static HFONT hfont;
        ...
        hfont = (HFONT)GetStockObject( DEFAULT_GUI_FONT );

```



```
SendMessage(hctrl, WM_SETFONT, (WPARAM)hfont,  
MAKELPARAM(TRUE, 0));
```

## Cambiar colores en combo box

Al tratarse de controles híbridos, formados mediante controles más simples, para cambiar el color de los controles combo box deberemos responder a varios de los mensajes de cambio de color, según la zona que nos interese personalizar.

Así, respondiendo al mensaje [WM\\_CTLCOLORLISTBOX](#), podremos modificar el color de fondo y el del texto de la parte de la lista.

Mediante el mensaje [WM\\_CTLCOLOREDIT](#), podremos modificar el color de fondo y del texto de la parte correspondiente al control de edición o estática (según el tipo de combo box).

Por supuesto, en cada caso deberemos retornar con un manipulador de pincel con el color usado para el fondo.

```
static HBRUSH pincel, pincel2;  
...  
case WM_CREATE:  
...  
    pincel = CreateSolidBrush( RGB(0,255,0) );  
    pincel2 = CreateSolidBrush( RGB(0,255,255) );  
...  
case WM_CTLCOLOREDIT:  
    SetTextColor( (HDC)wParam, RGB(255,255,0) );  
    SetBkColor( (HDC)wParam, RGB(0,255,255) );  
    return (LRESULT)pincel2;  
case WM_CTLCOLORLISTBOX:  
    SetTextColor( (HDC)wParam, RGB(255,0,0) );  
    SetBkColor( (HDC)wParam, RGB(0,255,0) );  
    return (LRESULT)pincel;  
...  
case WM_DESTROY:  
...  
    DeleteObject(pincel);  
    DeleteObject(pincel2);  
...
```

## Mensajes de notificación

Entre los mensajes de notificación de los controles combo box, los hay relativos a la parte del control de edición y a la parte de la lista.

Como en todos los casos, los mensajes de notificación se reciben a través de un mensaje [WM\\_COMMAND](#). En la palabra de menor peso del parámetro *wParam* se envía el identificador del control. En el parámetro *lParam* se envía el manipulador del control y el código del mensaje de notificación en la palabra de mayor peso de *wParam*.

### Nota:

En el API de Windows 3.x el código del mensaje de notificación se envía en el parámetro *lParam*. Hay que tener esto en cuenta si se intenta portar código entre estas plataformas.

## Cambio en selección de lista

El mensaje de notificación [CBN\\_SELCHANGE](#) se envía a la ventana padre cada vez que la selección en el list box de un combo box vaya a cambiar por una acción del usuario.

## Validar selección

El mensaje de notificación [CBN\\_SELENDOK](#) se envía a la ventana padre si después de hacer una selección se cierra la lista. El mensaje [CBN\\_SELENDCANCEL](#) se envía si después de una

selección, la lista no se pliega, sino que se cierra el cuadro de diálogo o el foco pasa a otro control.

Estos mensajes sólo se envían a combo boxes que no tengan el estilo [CBS\\_SIMPLE](#), ya que en el caso de los combo boxes simples, la lista no puede ser cerrada.

## **Despliegue de lista**

El mensaje de notificación [CBN\\_DROPDOWN](#) se envía cada vez que la lista de un combo box es desplegada. De forma simétrica, se envía un mensaje [CBN\\_CLOSEUP](#) cada vez que la lista se pliega de nuevo.

## **Doble clic**

Cada vez que el usuario hace doble clic sobre uno de los elementos del list box de un combo box, se envía un mensaje de notificación [CBN\\_DBLCLK](#) a su ventana padre.

## **Falta espacio**

Si no es posible conseguir memoria para completar una operación sobre el list box, se envía un mensaje de notificación [CBN\\_ERRSPACE](#).

## **Modificación**

Cada vez que el usuario modifica el texto de la parte del control de edición, primero se actualiza el contenido del control en pantalla, y a continuación se genera un mensaje [CBN\\_EDITCHANGE](#).

## **Actualización**

Cada vez que el usuario modifica el texto de la parte del control de edición, y antes de que este nuevo texto se muestre en pantalla,

Windows envía un mensaje [CBN\\_EDITUPDATE](#).

## Pérdida y recuperación de foco

Cada vez que el usuario selecciona otro control se envía un mensaje de notificación [CBN\\_KILLFOCUS](#).

Cuando el usuario selecciona un control list box, se envía un mensaje de notificación [CBN\\_SETFOCUS](#).

## Otros estilos para combo box

Entre los estilos también hay es evidente que estos controles son una mezcla de un control de edición o estático con un control de lista. Esto se ve en que hay estilo que sólo afectan a una de las partes o a las dos.

### Estilos para la parte de edición

- [CBS\\_AUTOHSCROLL](#): cuando el texto que se introduce en la parte del control de edición de un combo box no cabe en el área destinada a mostrarlo, se desplaza automáticamente a la derecha. Si no se activa este estilo, sólo podrá introducirse el texto que cabe dentro de los límites del cuadro de edición.
- [CBS\\_LOWERCASE](#): cualquier texto introducido en el control edit del combo box se convierte a minúsculas.
- [CBS\\_UPPERCASE](#): cualquier texto introducido en el control edit del combo box se convierte a mayúsculas.

### Estilos para la lista

Ya hemos visto antes algunos de estos estilos, pero comentaremos todos de nuevo:

- [CBS\\_DISABLENOSCROLL](#): se muestra siempre la barra de scroll vertical del list box, aunque aparece deshabilitada cuando

no hay suficientes elementos para llenar la lista. Si no se especifica este estilo, la barra de scroll se oculta, en lugar de deshabilitarse, cuando no hay suficientes elementos en la lista.

- **CBS\_HASSTRINGS**: como en el caso de los controles list box, este estilo está siempre activo en combo boxes que no tengan el estilo owner-drawn (controles que son actualizados por la ventana padre). Este estilo indica que el control contiene elementos que son cadenas de caracteres. El combo box es el encargado de mantener la memoria y las direcciones de las cadenas, de modo que la aplicación puede usar el mensaje **CB\_GETLBTEXT** para recuperar el texto de un elemento en particular.
- **CBS\_NOINTEGRALHEIGHT**: especifica que el tamaño de la lista del combo box es exactamente el indicado por la aplicación cuando se creó. Normalmente, Windows cambia el tamaño del combo box para que se muestren líneas completas, y no se corte el texto correspondiente a la última línea de la lista.
- **CBS\_SORT**: ordena alfabéticamente, de forma automática las cadenas introducidas en la lista.

## Ejemplo 71

### Mensajes correspondientes a la lista

La lista del combo box normalmente se inicializa por la aplicación, y el usuario puede elegir uno de los elementos de la lista, y salvo en el estilo de lista desplegable, también se podrán introducir valores que no estén en la lista.

### Añadir ítems

Mediante el envío de un mensaje **CB\_ADDSTRING** podemos añadir cadenas a la lista de un combo box, la dirección de la cadena se envía en el parámetro *lParam*.

Si el combo box tiene el estilo **CBS\_SORT**, la nueva cadena se inserta en la posición adecuada para mantener el orden alfabético entre los elementos de la lista.

```
sprintf(cad, "NUEVA CADENA");  
SendMessage(hctrl, CB_ADDSTRING, 0, (LPARAM)cad);
```

El mensaje **CB\_INSERTSTRING** nos permite insertar cadenas en la lista en una posición determinada, independientemente de que el combo box tenga el estilo **CBS\_SORT**, la cadena siempre se insertará en la posición especificada por el parámetro *wParam*. La cadena se indica en el parámetro *lParam*.

```
/* Insertar un ítem antes del seleccionado actualmente */  
strcpy(cad, "CADENA INSERTADA");  
SendMessage(hctrl, CB_INSERTSTRING, (WPARAM)3,  
(LPARAM)cad);
```

## Recuperar información

Los mensajes **CB\_GETLBTEXT** y **CB\_GETLBTEXTLEN** nos sirven para leer cadenas desde la lista de un combo box. Para el primero se indica en el parámetro *wParam* el índice del ítem a recuperar, y en *lParam* la dirección del buffer donde se almacena la cadena leída. El segundo mensaje nos sirve para obtener la longitud de la cadena de un ítem, indicado mediante su índice en el parámetro *wParam*.

```
/* Obtener cadena seleccionada */  
int i, l;  
char *cad;  
...  
i = SendMessage(hctrl, CB_GETCURSEL, 0, 0);  
l = SendMessage(hctrl, CB_GETLBTEXTLEN, (WPARAM)i,
```

```
(LPARAM) cad);  
    cad = (char*)malloc(l+1);  
    SendMessage(hctrl, CB_GETLBTEXT (WPARAM)i, (LPARAM) cad);
```

El mensaje [CB\\_GETCOUNT](#) no tiene parámetros, y sirve para obtener el número de elementos que contiene la lista de un combo box.

```
/* Obtener número de ítems */  
int i;  
...  
i = SendMessage(hctrl, CB_GETCOUNT, 0, 0);  
sprintf(cad, "Número de ítems: %d", i);  
MessageBox(hwnd, cad, "Combo Box", MB_OK);
```

Para obtener el índice del ítem actualmente seleccionado en la lista de un combo box, se usa el mensaje [CB\\_GETCURSEL](#). Este mensaje no precisa parámetros.

```
int i;  
...  
i = SendMessage(hctrl, CB_GETCURSEL, 0, 0);
```

En el caso de controles combo box con el estilo [CBS\\_DROPDOWN](#), en general nos interesará recuperar el texto que aparezca en el control de edición. Ya que el usuario puede escribir un texto que no aparece en la lista, recuperar el texto del control de edición asegura que siempre recuperamos el valor introducido por el usuario.

Para recuperar ese valor usaremos el mensaje [WM\\_GETTEXT](#):

```
SendDlgItemMessage(hwnd, ID_COMBO, WM_GETTEXT, 128,  
(LPARAM) cad);
```

O bien:

```
hctrl = GetDlgItem(hwnd, ID_COMBO);  
SendMessage(hctrl, WM_GETTEXT, 128, (LPARAM)cad);
```

Recordemos que en parámetro *wParam* indicaremos la longitud máxima de la cadena a recuperar y en *lParam* pasaremos un puntero a char, es decir, la dirección donde almacenaremos la cadena.

## Cambiar la selección

El mensaje **CB\_SETCURSEL** nos permite seleccionar un ítem, indicado por el parámetro *wParam*. Además, se elimina cualquier selección previa, y el contenido de la lista se desplaza, si es necesario, para mostrar la nueva cadena seleccionada.

```
/* Seleccionar ítem siguiente al actual */  
int i;  
...  
i = SendMessage(hctrl, CB_GETCURSEL, 0, 0);  
SendMessage(hctrl, CB_SETCURSEL, (LPARAM)i+1, 0);
```

El mensaje **CB\_SELECTSTRING** sirve para seleccionar una cadena determinada. En el parámetro *wParam* se envía el índice en que debe comenzar la búsqueda y en *lParam* la dirección de la cadena a buscar.

```
/* Seleccionar primera cadena que empieza por "E",  
después del 6º ítem */  
int i=6;  
...  
SendMessage(hctrl, CB_SELECTSTRING, (LPARAM)i,  
(LPARAM) "E");
```



## Buscar ítems

El mensaje **CB\_FINDSTRING** nos permite buscar una cadena que coincida con el prefijo especificado en el parámetro *lParam*, a partir del índice indicado en *wParam*:

```
/* Seleccionar la primera cadena que empiece por "EN" */
int i;
...
i = SendMessage(hwnd, CB_FINDSTRING, (WPARAM)-1,
(LPARAM) "EN");
SendMessage(hwnd, CB_SETCURSEL, (WPARAM)i, 0);
```

El mensaje **CB\_FINDSTRINGEXACT** es parecido, pero no usa el parámetro *lParam* como un prefijo, sino que busca una cadena que coincida exactamente con ese parámetro.

```
/* Seleccionar la cadena igual a "Enero" */
int i;
...
i = SendMessage(hwnd, CB_FINDSTRINGEXACT, (WPARAM)-1,
(LPARAM) "Enero");
SendMessage(hwnd, CB_SETCURSEL, (WPARAM)i, 0);
```

En cualquiera de los dos casos, si no se encuentra la cadena buscada, el valor de retorno es **CB\_ERR**.

## Borrar ítems

Para eliminar ítems se usa el mensaje **CB\_DELETESTRING**, en el que indicaremos en el parámetro *wParam* el valor del índice a eliminar.

```
/* Eliminar cadena actualmente seleccionada */
int i;
```

```
...  
    i = SendMessage(hctrl, CB_GETCURRESEL, 0, 0);  
    SendMessage(hctrl, CB_DELETESTRING, (WPARAM)i, 0);
```

Mediante el mensaje **CB\_RESETCONTENT**, sin parámetros, podemos vaciar una lista de un combo box por completo.

```
/* Vaciar list box */  
SendMessage(hctrl, CB_RESETCONTENT, 0, 0);
```

## Otros mensajes

El mensaje **CB\_GETTOPINDEX** sirve para recuperar el índice del primer ítem visible de la lista de un combo box. Este mensaje no tiene parámetros.

De forma simétrica, el mensaje **CB\_SETTOPINDEX** sirve para asegurar que un determinado ítem estará en la parte visible de la lista de un combo box. En el parámetro *wParam* se indica el índice del ítem que queremos que sea visible.

Mediante el mensaje **CB\_SHOWDROPDOWN** podemos mostrar u ocultar la lista desplegable asociada a un combo box con el estilo **CBS\_DROPDOWN** o **CBS\_DROPDOWNLIST**.

También podemos obtener el estado de la lista desplegable mediante el mensaje **CB\_GETDROPPEDSTATE**. Este mensaje no tiene parámetros, y el valor de retorno indica si la lista está desplegada (**TRUE**) o plegada (**FALSE**).

## Ejemplo 72

### El dato del ítem

Ya sabemos que cada ítem tiene asociado un índice y una cadena. Pero también tiene asociado un dato entero de 32 bits: el

ítem data, o dato de ítem.

A cada ítem le podemos asignar un valor entero mediante el mensaje `CB_SETITEMDATA`, y recuperarlo mediante `CB_GETITEMDATA`.

Podemos aprovechar que el valor de retorno del mensaje `CB_ADDSTRING` es el índice del ítem insertado, y usar ese valor en el mensaje `CB_SETITEMDATA`, para asignar el valor del índice en el array:

```
void IniciarLista(HWND hctrl)
{
    char* mes[] = {"Enero", "Febrero", "Marzo", "Abril",
"Mayo",
    "Junio", "Julio", "Agosto", "Septiembre", "Octubre",
    "Noviembre", "Diciembre"};
    int i;
    int actual;

    for(i = 0; i < 12; i++) {
        actual = SendMessage(hctrl, CB_ADDSTRING, 0,
(LPARAM)mes[i]);
        SendMessage(hctrl, CB_SETITEMDATA, (LPARAM)actual, i);
    }
}
```

Ya veremos que el dato de ítem tiene otras utilidades, pero en muchos casos nos proporciona una forma útil de almacenar un dato relativo a un ítem. Al tratarse de un entero de 32 bits también puede contener punteros.

## Interfaces de usuario

Existen dos comportamientos diferentes de los combo boxes con listas desplegables (con el estilo `CBS_DROPDOWN` o `CBS_DROPDOWNLIST`), con respecto al teclado. Cada uno de estos comportamientos viene definido por un interfaz.

En el interfaz por defecto, la tecla F4 despliega u oculta la lista desplegable, y las teclas de flecha arriba y abajo nos permiten desplazarnos a través de las distintas opciones.

En el interfaz extendido la tecla F4 no tiene ninguna función, y la lista se despliega tan pronto como se pulsa una de las teclas de flecha arriba o abajo.

Las flechas funcionan tanto si la lista está desplegada como si no.

Para cambiar el interfaz de un control combo box se usa el mensaje `CB_SETEXTENDEDUI`, si se usa el valor `FALSE` en el parámetro *wParam*, se activa el interfaz por defecto, y con el valor `TRUE` se activa el interfaz extendido.

Mediante el mensaje `CB_GETEXTENDEDUI` se puede obtener el valor actual del interfaz para un control combo box determinado. Este mensaje no tiene parámetros, y el valor de retorno indica el interfaz asociado al control. `TRUE` si es el extendido y `FALSE` si es el interfaz por defecto.

## Funciones para ficheros y directorios

Como vimos con los list boxes, en los combo boxes también es posible iniciar la lista usando los nombres de ficheros de una unidad de disco o un directorio.

La función `DlgDirListComboBox` nos permite iniciar el contenido de una lista asociada a un combo box a partir de los ficheros, carpetas, unidades de disco, etc.

Esta función necesita cinco parámetros. El primero es un manipulador de la ventana o diálogo que contiene el combo box que vamos a inicializar. El segundo es un puntero a una cadena con el camino del directorio a mostrar. El tercer parámetro es el identificador del combo box. El cuarto el identificador de un control estático, que se usa para mostrar el camino actualmente mostrado en el combo box. El último parámetro nos permite seleccionar el tipo de entradas que se mostrarán.

Mediante este último parámetro podemos restringir el tipo de entradas, impidiendo o permitiendo que se muestren directorios o unidades de almacenamiento, o limitando los atributos de los ficheros y directorios a mostrar.

Ya hemos dicho que se necesita un control estático.

```
    HWND hestatico;  
    ...  
        hestatico = CreateWindowEx(  
            0,  
            "STATIC",          /* Nombre de la clase */  
            "",                /* Texto del título, no tiene  
*/  
            WS_CHILD | WS_VISIBLE | WS_BORDER |  
WS_TABSTOP, /* Estilo */  
            9, 4,              /* Posición */  
            344, 18,          /* Tamaño */  
            hwnd,              /* Ventana padre */  
            (HMENU)ID_TITULO, /* Identificador del control  
*/  
            hInstance,         /* Instancia */  
            NULL);             /* Sin datos de creación de  
ventana */  
        SendMessage(hestatico, WM_SETFONT, (WPARAM)hfont,  
MAKELPARAM(TRUE, 0));
```

Por supuesto, podemos usar los comodines '\*' y '?' para los nombres de fichero:

```
    ...  
    IniciarCombo(hwnd, "*.c");  
    ...  
  
void IniciarCombo(HWND hwnd, char* p)  
{  
    char path[512];  
  
    strcpy(path, p);  
  
    DlgDirListComboBox(  

```

```

    hwnd,          /* manipulador de cuadro de diálogo con
list box */
    path,          /* puntero a cadena de camino o nombre
de fichero */
    ID_COMBO,      /* identificador de list box
*/
    ID_TITULO,     /* identificador de control estático
*/
    DDL_DIRECTORY | DDL_DRIVES /* atributos de ficheros a
mostrar */
);
}

```

Por otra parte, la función [DlgDirSelectComboBoxEx](#) nos permite leer la selección actual de un combo box inicializado mediante la función [DlgDirListComboBox](#). Si el valor de retorno de esta función es distinto de cero, la selección actual es un directorio o unidad de almacenamiento, por lo que será posible hacer un cambio de directorio. Si el valor de retorno es cero, se trata de un fichero.

Aprovecharemos esto para navegar a lo largo de los discos de nuestro ordenador, para lo que responderemos al mensaje de notificación [CBN\\_DBLCLK](#), cambiando a la nueva ubicación o mostrando el nombre del fichero seleccionado:

```

    case WM_COMMAND:
        switch(LOWORD(wParam)) {
            case ID_COMBO:
                switch(HIWORD(wParam)) {
                    case CBN_CLOSEUP:
                        if(DlgDirSelectComboBoxEx(hwnd, cad,
512, ID_COMBO)) {
                            strcat(cad, "*.c");
                            IniciarCombo(hwnd, cad);
                        } else
                            {MessageBox(hwnd, cad, "Fichero
seleccionado", MB_OK);
                                break;
                            ...

```

También existe un mensaje relacionado con este tema.

El mensaje [CB\\_DIR](#) tiene un uso equivalente a la función [DlgDirListComboBox](#). En el parámetro *wParam* se indican los atributos de los ficheros a mostrar, así como si se deben mostrar directorios y unidades de almacenamiento. En el parámetro *lParam* se suministra el nombre de fichero, que puede tener comodines) o el camino de los ficheros a insertar.

Por ejemplo, podemos añadir los ficheros de cabecera al contenido del combo box, de modo que se muestren los ficheros fuente en c y los de cabecera:

```
void IniciarLista(HWND hwnd, char* p)
{
    char path[512];

    strcpy(path, p);

    DlgDirComboBox(hwnd, path, ID_LISTA, ID_TITULO,
        DDL_DIRECTORY | DDL_DRIVES);

    strcpy(path, "*.h");
    SendMessage(GetDlgItem(hwnd, ID_LISTA), CB_DIR,
        (LPARAM)0, (LPARAM)path);
}
```

## Juegos de caracteres

Disponemos de un estilo específico para controles combo box que contienen nombres de ficheros, se trata del estilo [CBS\\_OEMCONVERT](#). Cuando un control tiene este estilo, las cadenas insertadas se convierten desde juego de caracteres de Windows al juego de caracteres OEM y después se vuelve a convertir al juego de caracteres de Windows. Esto asegura una conversión correcta si la aplicación llama a la función [CharToOem](#) para convertir una cadena Windows del combo box a una cadena OEM. Este sólo se puede aplicar a controles con el estilo [CBS\\_SIMPLE](#) o [CBS\\_DROPDOWN](#).

## Procesar CBN\_CLOSEUP

Es aconsejable procesar el mensaje de notificación [CBN\\_CLOSEUP](#) cuando se trabaja con combo boxes que contienen directorios o ficheros, sobre todo cuando la elección de una de las opciones requiera algún tipo de proceso, por ejemplo, al cambiar de directorio se deberá limpiar la lista y volver a generarla. En estos casos no sería lógico procesar el mensaje de notificación [CBN\\_SELCHANGE](#).

## Selección actual

Es el ítem de la lista del combo box seleccionado por el usuario. Su texto se copia en el campo de selección, tanto si se trata de un control de edición o de uno estático. Salvo en el caso de la lista desplegable, existe otra forma de introducir datos en un combo box: teclearlos en el control edit.

Ya hemos visto que es posible recuperar el valor de la selección actual mediante un mensaje [CB\\_GETCURSEL](#), o cambiarlo mediante un mensaje [CB\\_SETCURSEL](#) o [CB\\_SELECTSTRING](#).

También hemos visto que existe un mensaje de notificación [CBN\\_SELCHANGE](#) que se envía a la ventana padre del combo box cada vez que el usuario cambia la selección actual. Hay que tener en cuenta que este mensaje de notificación no se envía si la selección actual se modifica directamente usando un mensaje [CB\\_SETCURSEL](#).

## Ejemplo 73

En este ejemplo hemos usado algunas funciones y estructuras del API, relacionadas con información sobre ficheros y manejo de tiempos, que no hemos comentado con anterioridad:



- [GetFileAttributesEx](#) sirve para obtener información sobre un fichero, sus atributos, fechas de creación, modificación y último acceso y tamaño.
- [WIN32\\_FILE\\_ATTRIBUTE\\_DATA](#) es la estructura usada por la función anterior para devolver los datos de un fichero.
- [FileTimeToSystemTime](#) las fechas devueltas por la función anterior están en formato de fecha de fichero, esta función permite traducir ese formato a formato de fecha de sistema, que es más manejable.
- [FILETIME](#) estructura que almacena una fecha en forma de número entero.
- [SYSTEMTIME](#) estructura que almacena una fecha en forma de campos individuales.

Veremos todos estos conceptos en capítulos futuros dedicados específicamente a ello.

## El control de edición

El campo de selección de un combo box puede ser un control de edición o un control estático. El la parte que contiene el valor de la selección actual o el texto introducido por el usuario.

Podemos usar el mensaje [WM\\_GETTEXT](#) para obtener el texto que contiene actualmente, o el mensaje [WM\\_SETTEXT](#) para modificarlo.

También podemos usar el mensaje [CB\\_GETEDITSEL](#) para obtener las posiciones de inicio y final de la selección de texto actual, si existe; o usar el mensaje [CB\\_SETEDITSEL](#) para seleccionar parte del texto dentro del control de edición.

Si se usa el estilo [CBS\\_AUTOHSCROLL](#) es posible introducir más texto del que se puede visualizar en la anchura del control, en caso contrario la cantidad de texto que es posible introducir estará limitada por esa anchura.

También se puede limitar el número de caracteres que el usuario puede teclear mediante el mensaje [CB\\_LIMITTEXT](#).

Hay dos mensajes de notificación relacionados con el control de edición. Cuando el se modifica el contenido del control de edición, la ventana padre recibe primero el mensaje de notificación [CBN\\_EDITUPDATE](#), para indicar que el texto se ha alterado. Después de que el texto se ha mostrado, Windows envía el mensaje [CBN\\_EDITCHANGE](#).

## Actualizaciones de gran número de ítems

Hay dos posibles situaciones de potencialmente peligrosas en las las actualizaciones que afecten a muchos ítems en un combo box.

Por una parte, el proceso puede requerir una cantidad importante de memoria, cuando se añaden muchos ítems.

Por otra parte, el proceso puede requerir mucho tiempo, ya sea porque se deben añadir muchos ítems o porque se deben hacer muchas modificaciones que impliquen el borrado e inserción de ítems.

## Optimizar la memoria

En versiones de Windows anteriores al uso de la memoria virtual, era necesario tener en cuenta la memoria disponible antes de insertar un gran número de ítems en un combo box. Para eso se usaba el mensaje [CB\\_INITSTORAGE](#), en el que indicamos en el parámetro wParam el número de ítems a añadir, y en el parámetro lParam la cantidad de memoria estimada necesaria para acomodar esos ítems.

```
/* Prepararse para insertar 10000 ítems de
32 bytes por ítem, aproximadamente */
SendMessage(hwnd, CB_INITSTORAGE, 10000, 320000);
IniciarLista(hwnd);
```

No es necesario ser demasiado preciso con la cantidad de memoria requerida, se trata sólo de una estimación, si nos quedamos cortos, los ítems que no quepan se insertarán del modo normal. Si nos quedamos largos, la memoria sobrante se podrá aprovechar en nuevas inserciones.

Este mensaje sólo es necesario en Windows 95, en NT no nos preocupa la memoria necesaria para almacenar los ítems, ya que el modelo de memoria virtual dispone de una cantidad prácticamente ilimitada.

## **Optimizar el tiempo**

El problema del tiempo sí es importante. Cada vez que se añade o elimina un ítem, el combo box intenta actualizar la pantalla para reflejar los cambios, al menos en combo boxes con el estilo `CBS_SIMPLE` donde la lista está permanentemente desplegada. Esto, cuando los cambios son muy numerosos, hará que aparentemente la aplicación no responda, y que el tiempo invertido en las actualizaciones sea mayor del necesario.

En el caso de los controles combo box no disponemos de un estilo equivalente al estilo `LBS_NOREDRAW` de los list box, que inhibe las actualizaciones de la lista. En el caso de los combo boxes, cuando puedan contener muchos valores en la lista, será mejor usar otros estilos diferentes de `CBS_SIMPLE`.

## **Aspectos gráficos del combo box**

En cuanto al aspecto gráfico del combo box tenemos otras opciones que podemos controlar.

### **Ajustar la anchura de un combo box**

Por una parte, ya vimos que podemos añadir una barra de desplazamiento horizontal creando nuestro combo box con el estilo

[WS\\_HSCROLL](#). Puede ser útil si la anchura de los ítems sobrepasa la del list box.

Sin embargo, usar este estilo no asegura que la barra de desplazamiento sea mostrada. Para que la barra aparezca hay que ajustar la extensión horizontal del list box mediante un mensaje [CB\\_SETHORIZONTALEXTENT](#), indicando en el parámetro wParam la nueva extensión horizontal, en pixels.

Si la extensión horizontal es mayor que la anchura del combo box, se mostrará la barra de desplazamiento, en caso contrario la barra no aparecerá.

Esto nos plantea una duda, ¿cómo calcular la extensión necesaria según las longitudes de las cadenas contenidas en el combo box?

Bueno, podríamos hacerlo a ojo, pero esta técnica es arriesgada, ya que si nos quedamos cortos no será posible visualizar por completo algunos ítems.

Lo mejor es calcular la longitud de cada cadena al insertarla, y si es mayor que la extensión actual, actualizar el valor de la extensión. Para obtener el valor de la extensión actual se usa el mensaje [CB\\_GETHORIZONTALEXTENT](#).

Claro que esto plantea un problema si se eliminan ítems, ya que nos obligaría a calcular las longitudes de todas las cadenas que quedan en el combo box. Sin embargo, podemos ignorar estos casos, y mantener la extensión, ya que la visibilidad de todos los ítems está asegurada.

Para calcular la longitud de una cadena en pixes, vimos en el [capítulo 24](#), que podemos usar la función [GetTextExtentPoint32](#), por ejemplo, en la siguiente función:

```
int CalculaLongitud(HWND hwnd, char *cad)
{
    HDC hdc;
    SIZE tam;
    HFONT hfont;
```

```

hfont = (HFONT)GetStockObject( DEFAULT_GUI_FONT );
hdc = GetDC(hwnd);
SelectObject(hdc, hfont);
GetTextExtentPoint32(hdc, cad, strlen(cad), &tam);
/*LPtoDP(hdc, (POINT *)&tam, 1);*/
ReleaseDC(hwnd, hdc);

return tam.cx;
}

```

Para que el cálculo sea correcto debemos seleccionar en el DC la misma fuente que usamos en el combo box. Además, habría que tener en cuenta que la función [GetTextExtentPoint32](#) devuelve el tamaño de la cadena en unidades lógicas, y en rigor habría que convertir esos valores a unidades de dispositivo. Pero esto es innecesario, ya que en un control no se realiza ninguna proyección.

Así, cada vez que insertemos un ítem en el combo box, deberemos comprobar si resulta ser el más largo:

```

char item[300];
int x;
int eActual;

eActual = SendMessage(hlista, CB_GETHORIZONTALEXTENT, 0, 0);

strcpy(item, "Ítem de una anchura tal que no cabe en "
        "el combo box que hemos definido, o al menos no debería caber, "
        "si las cosas salen tal y como las hemos calculado, claro.");

x = CalculaLongitud(hlista, cad);
if(x > eActual) eActual = x;
SendMessage(hlista, CB_ADDSTRING, 0, (LPARAM)cad);
SendMessage(hlista, CB_SETHORIZONTALEXTENT, eActual, 0);

```

## Ajustar la altura de los ítems

Por defecto, la altura de los ítems se calcula en función de la fuente asignada al list box. Podemos obtener el valor de la altura del ítem mediante el mensaje `CB_GETITEMHEIGHT`. Si se trata de un combo box con un estilo owner-draw cada ítem puede tener una altura diferente, y se puede especificar el índice del ítem en el parámetro `wParam`. En los combo box normales, el valor de `wParam` debe ser cero.

```
h = SendMessage(hctrl, CB_GETITEMHEIGHT, 0, 0);
```

Para modificar la altura de un ítem se usa el mensaje `CB_SETITEMHEIGHT`, en el caso de combo boxes con un estilo owner-draw se puede asignar una altura diferente a cada ítem. En ese caso, especificaremos el índice del ítem en el parámetro `wParam`, y la altura deseada en la palabra de menor peso del parámetro `lParam`, usando la macro `MAKELPARAM`. Veremos esto con más detalle al estudiar los estilos owner-draw.

```
SendMessage(hctrl, CB_SETITEMHEIGHT, 0,  
MAKELPARAM(30, 0));  
InvalidateRect(hctrl, NULL, TRUE);
```

## Localizaciones

Ya hemos visto que en los controles combo box los ítems se muestran por orden alfabético, al menos en los que hemos usado hasta ahora. Pero el orden alfabético no es algo universal, y puede cambiar dependiendo del idioma.

Generalmente esto no nos preocupará, ya que el idioma usado para elegir el orden se toma del propio sistema. Sin embargo, puede haber casos en que nos interese modificar o conocer el idioma usado en un combo box.

Para obtener el valor de la localización actual se usa el mensaje `CB_GETLOCALE`. El valor de retorno es un entero de 32 bits, en el que la palabra de menor peso contiene el código de país, y el de mayor peso el del lenguaje, este último a su vez, se compone de un identificador de lenguaje primario y un identificador de sublenguaje.

Se pueden usar las macros `PRIMARYLANGID` y `SUBLANGID` para obtener el identificador de lenguaje primario y el de sublenguaje, respectivamente.

```
int i;
char cad[120];
...
i = SendMessage(hwnd, CB_GETLOCALE, 0, 0);
sprintf(cad, "País %d, id lenguaje primario %d, "
        "id de sublenguaje %d",
        HIWORD(i), PRIMARYLANGID(LOWORD(i)),
        SUBLANGID(LOWORD(i)));
MessageBox(hwnd, cad, "Localización", MB_OK);
```

También podemos modificar la localización actual mediante un mensaje `CB_SETLOCALE`, indicando en el parámetro `wParam` el nuevo valor de localización. Podemos crear uno de estos valores mediante las macros `MAKELCID` y `MAKELANGID`:

```
SendMessage(hwnd, CB_SETLOCALE,
            MAKELCID(MAKELANGID(LANG_SPANISH,
                                SUBLANG_SPANISH),
                    SSORT_DEFAULT), 0);
```

La macro `MAKELCID` crea un identificador de localización a partir de un identificador de lenguaje y una constante que debe ser `SSORT_DEFAULT`.

La macro `MAKELANGID` crea un identificador de lenguaje a partir de un identificador de lenguaje primario y de un identificador de sublenguaje.

## Combo boxes owner draw

El funcionamiento de los controles combo box owner-draw es muy similar al de los controles list box. Mucho de lo que se comentó para estos controles se aplica igualmente a los combo box.

De modo que también existen dos estilos distintos owner-draw que se pueden aplicar a los controles combo box [CBS\\_OWNERDRAWFIXED](#) y [CBS\\_OWNERDRAWVARIABLE](#).

El primero define controles combo box owner-draw en los que la altura de todos los ítems es la misma. En el segundo caso, las alturas de cada ítem pueden ser diferentes.

Como también pasa en los controles list box, en los combo box con estilos owner-draw tampoco se activa por defecto el estilo [CBS\\_HASSTRINGS](#). Si estamos personalizando nuestros controles, lo más probable que éste no contenga cadenas, o al menos, no sólo cadenas.

Sin embargo, es posible que aún tratándose de un combo box con un estilo owner-draw, nuestro control contenga cadenas. En ese caso podemos activar el estilo [CBS\\_HASSTRINGS](#), sobre todo si queremos que los ítems se muestren por orden alfabético.

```
hctrl = CreateWindowEx(
    0,
    "COMBOBOX"           /* Nombre de la clase */
    "",                  /* Texto del título */
    CBS_HASSTRINGS | CBS_OWNERDRAWVARIABLE |
    CBS_DROPDOWNLIST | CBS_SORT |
    WS_CHILD | WS_VISIBLE | WS_BORDER |
    WS_TABSTOP, /* Estilo */
    9, 19,             /* Posición */
    320, 250,          /* Tamaño */
    hwnd,              /* Ventana padre */
    (HMENU) ID_COMBO, /* Identificador del control */
    /*
    hInstance,          /* Instancia */
    NULL);              /* Sin datos de creación de
    ventana */
```



Si no activamos el estilo [CBS\\_HASSTRINGS](#), el valor que usemos al insertar el ítem será almacenado en el dato del ítem de 32 bits.

```
void IniciarCombo(HWND hctrl)
{
    int i;

    for(i = 0; i < 10; i++)
        SendMessage(hctrl, CB_ADDSTRING, 0, i);
}
```

## Combo box owner-draw de altura fija

La ventana propietaria del control recibirá el mensaje [WM\\_MEASUREITEM](#) cuando el control combo box sea creado.

En el parámetro *lParam* recibiremos un puntero a una estructura [MEASUREITEMSTRUCT](#) que contiene las dimensiones del control.

En el parámetro *wParam* recibiremos el valor del identificador del control, o lo que es lo mismo, el valor del miembro *CtlID* de la estructura [MEASUREITEMSTRUCT](#) apuntada por el parámetro *lParam*. Este valor identifica el control del que procede el mensaje [WM\\_MEASUREITEM](#).

Tengamos en cuenta que pueden existir varios controles con el estilo owner-draw, y no tienen por qué ser necesariamente del tipo combo box. Si este valor es cero, el mensaje fue enviado por un menú. Si el valor es distinto de cero, el mensaje fue enviado por un combobox o por un listbox.

Nuestra aplicación debe rellenar de forma adecuada la estructura [MEASUREITEMSTRUCT](#) apuntada por el parámetro *lParam* regresar. De este modo se indica al sistema operativo qué dimensiones tiene el control.

El mensaje [WM\\_MEASUREITEM](#) se envía a la ventana propietaria del combo box antes de enviar el mensaje [WM\\_INITDIALOG](#) o [WM\\_CREATE](#), de modo que en ese momento

Windows aún no ha determinado la altura y anchura de la fuente usada en el control.

Si se procesa este mensaje se debe retornar el valor **TRUE**.

```
switch(msg)                                /* manipulador del mensaje
*/
{
    case WM_CREATE:
        ...
    case WM_MEASUREITEM:
        lpmis = (LPMEASUREITEMSTRUCT) lParam;
        lpmis->itemHeight = 40;
        return TRUE;
    ...
}
```

## Combo box owner-draw de altura variable

En este caso, la ventana propietaria del control recibirá el mensaje **WM\_MEASUREITEM** cada vez que se inserte un nuevo ítem en el control combo box. Esto nos permitirá ajustar la altura de cada ítem con valores diferentes.

El proceso del mensaje es idéntico que con el estilo **CBS\_OWNERDRAWFIXED**. La diferencia es que este mensaje se enviará para cada ítem, y siempre después del mensaje **WM\_INITDIALOG** o **WM\_CREATE**.

## Dibujar cada ítem

Tanto en un caso como en el otro, Windows enviará un mensaje cada vez que se inserte un nuevo ítem, cuando el estado de un ítem cambie o cuando un ítem deba ser mostrado.

Esto se hace mediante un mensaje **WM\_DRAWITEM**. En el parámetro wParam recibiremos el identificador del control del que procede el mensaje, o cero si es un menú. En el parámetro lParam

recibiremos un puntero a una estructura **DRAWITEMSTRUCT**, que contiene toda la información relativa al ítem que hay que mostrar.

Si se procesa este mensaje hay que retornar el valor **TRUE**.

Procesar este mensaje puede ser un proceso bastante complejo, ya que el estado de un ítem puede tomar varios valores diferentes, y seguramente, cuando decidimos crear un control owner-draw es porque queremos hacer algo especial.

La estructura **DRAWITEMSTRUCT** tiene esta forma:

```
typedef struct tagDRAWITEMSTRUCT {    // dis
    UINT    CtlType;
    UINT    CtlID;
    UINT    itemID;
    UINT    itemAction;
    UINT    itemState;
    HWND    hwndItem;
    HDC     hDC;
    RECT    rcItem;
    DWORD   itemData;
} DRAWITEMSTRUCT;
```

En nuestro caso, *CtlType* tendrá el valor **ODT\_COMBOBOX**, pero tengamos en cuenta que habrá que discriminar este miembro si tenemos controles owner-draw de distintos tipos.

*CtlID* contiene el identificador del control, igual que el parámetro *wParam*.

*itemID* contiene el índice del ítem . Si el combo box está vacío, el valor será -1.

*itemAction* puede tener tres valores diferentes, que en ocasiones requerirán un tratamiento distinto por parte de nuestro programa:

- **ODA\_DRAWENTIRE** indica que el ítem debe ser dibujado por entero.
- **ODA\_FOCUS** indica que el control ha perdido o recuperado el foco. Para saber si se trata de uno u otro caso se debe comprobar el miembro *itemState*.

- **ODA\_SELECT** indica que el estado de selección del ítem ha cambiado. Para saber si el ítem está ahora seleccionado o no también se debe comprobar el miembro *itemState*.

*itemState* indica el estado del ítem. El valor puede ser uno o una combinación de los siguientes:

- **ODS\_COMBOBOXEDIT** se está dibujando el campo de selección (control edit) del combo box owner drawn.
- **ODS\_DEFAULT** se trata del ítem por defecto.
- **ODS\_DISABLED** el ítem está deshabilitado.
- **ODS\_FOCUS** el ítem tiene el foco.
- **ODS\_SELECTED** el ítem está seleccionado.

*hwndItem* contiene el manipulador de ventana del control.

*hDC* contiene el manipulador de contexto de dispositivo del control. Este valor nos será muy útil, ya que el proceso de este mensaje será en encargado de dibujar el ítem.

*rcItem* contiene un rectángulo que define el contorno del ítem que estamos dibujando. Además este rectángulo define una región de recorte, de modo que no podremos dibujar nada fuera de él.

*itemData* contiene el valor del 32 bits asociado al ítem.

Con esto tenemos toda la información necesaria para dibujar cada ítem, y nuestro programa será el responsable de diferenciar los distintos estados de cada uno.

```
case WM_DRAWITEM:
    lpdis = (LPDRAWITEMSTRUCT) lParam;
    if(lpdis->itemID == -1) { /* Se trata de un
menú, no hacer nada */
        break;
    }
    switch (lpdis->itemAction) {
        case ODA_SELECT:
        case ODA_DRAWENTIRE:
        case ODA_FOCUS:
            /* Borrar el contenido previo */
            FillRect(lpdis->hDC, &lpdis->rcItem, (HBRUSH)
```

```

(COLOR_WINDOW+1));
/* Obtener datos de las medidas de la fuente
*/
GetTextMetrics(lpdis->hDC, &tm);
/* Calcular la coordenada y para escribir el
texto de ítem */
y = (lpdis->rcItem.bottom + lpdis->rcItem.top
- tm.tmHeight) / 2;
/* Cada tipo de comida se muestra en un color
diferente */
if(comida[lpdis->itemData].tipo == 'p')
    SetTextColor(lpdis->hDC, RGB(0,128,0));
else
if(comida[lpdis->itemData].tipo == 'c')
    SetTextColor(lpdis->hDC, RGB(0,0,255));
else
if(comida[lpdis->itemData].tipo == 'b')
    SetTextColor(lpdis->hDC, RGB(255,0,0));
/* Mostrar el icono */
icono = LoadIcon(hInstance,
MAKEINTRESOURCE(Icono+lpdis->itemData));
DrawIcon(lpdis->hDC, 4, lpdis->rcItem.top+2,
icono);

DeleteObject(icono);
/* Mostrar el texto */
TextOut(lpdis->hDC, 42, y,
comida[lpdis->itemData].nombre,
strlen(comida[lpdis->itemData].nombre));
/* Si el ítem está seleccionado, trazar un
rectángulo negro alrededor */
if (lpdis->itemState & ODS_SELECTED) {
    SetTextColor(lpdis->hDC, RGB(0,0,0));
    DrawFocusRect(lpdis->hDC, &lpdis->rcItem);
}
}
break;

...

```

Este ejemplo usa la lista de países de ejemplos anteriores, hemos hecho que los países de más de 92391 km<sup>2</sup> se muestren en color verde, y el resto en azul.

Por supuesto, esta es una aplicación muy sencilla de un combo box owner-draw. Es posible personalizar tanto como queramos

estos controles, mostrando mapas de bits o cualquier gráfico que queramos.

## Otros mensajes para combo box con estilos owner-draw

Disponemos de otros mensajes destinados a controles owner-draw.

El mensaje `CB_GETITEMHEIGHT` se puede usar para obtener la altura de los ítems en un combo box owner-draw. Si el control tiene el estilo `CBS_OWNERDRAWFIXED` tanto el parámetro `lParam` como `wParam` deben ser cero. Si el control tiene el estilo `CBS_OWNERDRAWVARIABLE`, el parámetro `wParam` debe contener el índice del ítem cuya altura queremos recuperar.

De forma simétrica, disponemos del mensaje `CB_SETITEMHEIGHT` para ajustar la altura de los ítems. Si se trata de un control con el estilo `CBS_OWNERDRAWFIXED` debe indicarse cero para el parámetro `wParam`, y la altura se especifica en el parámetro `lParam`, para lo que será necesario usar la macro `MAKELPARAM`:

```
SendMessage(hwnd, CB_SETITEMHEIGHT, 0, MAKELPARAM(23, 0));
```

Si se trata de un control con el estilo `CBS_OWNERDRAWVARIABLE` procederemos del mismo modo, pero indicando en el parámetro `wParam` el índice del ítem cuya altura queremos modificar.

## El mensaje `WM_DELETEITEM`

Cuando se elimina un ítem de un combo box cuyo dato de ítem no sea nulo, en Windows 95; o para ítems pertenecientes a controles owner draw, en el caso de Windows NT, el sistema envía un mensaje `WM_DELETEITEM` al procedimiento de ventana de la

ventana propietaria del control. Concretamente, esto ocurre cuando se usan los mensajes `CB_DELETESTRING` o `CB_RESETCONTENT` o cuando el propio control es destruido.

Esto nos da una oportunidad de tomar ciertas decisiones o realizar ciertas tareas cuando algunos ítems concretos son eliminados.

En el parámetro `wParam` recibiremos el identificador del control en el que se ha eliminado el ítem. En el parámetro `lParam` recibiremos un puntero a una estructura `DELETEITEMSTRUCT`. Esta estructura está definida como:

```
typedef struct tagDELETEITEMSTRUCT { // ditms
    UINT CtlType;
    UINT CtlID;
    UINT itemID;
    HWND hwndItem;
    UINT itemData;
} DELETEITEMSTRUCT;
```

*CtlType* contiene el valor `ODT_COMBOBOX`.

*CtlId* contiene el valor del identificador del control.

*itemID* el valor del índice del ítem eliminado.

*hwndItem* el manipulador de ventana del control.

*itemData* el dato del ítem asignado al ítem eliminado.

## Dimensiones de la lista desplegable

Ya sólo quedan por comentar dos mensajes más relacionados con los combo boxes.

Los dos están relacionados con el tamaño de la lista desplegable, uno de ellos nos permite obtener el rectángulo correspondiente a esa lista, `CB_GETDROPPEDCONTROLRECT`, el parámetro `wParam` no se usa, y debe ser cero, el parámetro `lParam` se usa para pasar un puntero a una estructura `RECT` en la que se nos devolverán las coordenadas que definen ese rectángulo.

El otro mensaje es [CB\\_SETDROPPEDWIDTH](#), que nos permite modificar la anchura de la lista. La nueva anchura se especifica mediante el parámetro *wParam*, el parámetro *lParam* no se usa, y debe ser cero.

## Definición del orden

Por último, cuando un control combo box tiene el estilo [CBS\\_SORT](#), el procedimiento de ventana de la ventana propietaria del control recibe uno o varios mensajes [WM\\_COMPAREITEM](#) para determinar la posición de cada nuevo ítem insertado en el control.

Esto nos permite definir nuestro propio orden para los ítems en el control, en lugar de usar el orden alfabético por defecto.

El mensaje se puede recibir varias veces para cada ítem insertado, ya que generalmente no será suficiente una comparación para determinar el orden.

En el parámetro *wParam* recibiremos el identificador del control, y en *lParam* un puntero a una estructura [COMPAREITEMSTRUCT](#), con todos los datos necesarios para determinar el orden entre dos ítems del combo box. Esta estructura tiene esta definición:

```
typedef struct tagCOMPAREITEMSTRUCT { // cis
    UINT    CtlType;
    UINT    CtlID;
    HWND    hwndItem;
    UINT    itemID1;
    DWORD   itemData1;
    UINT    itemID2;
    DWORD   itemData2;
} COMPAREITEMSTRUCT;
```

*CtlType* contendrá el valor `ODT_COMBOBOX`.

*CtlID* el valor del identificador del control.

*hwndItem* el manipulador de ventana del control.

*itemID1* el índice del primer ítem a comparar.



*itemData1* el valor del dato del ítem del primer ítem a comparar.

*itemID2* el índice del segundo ítem a comparar.

*itemData2* el valor del dato del ítem del segundo ítem a comparar.

El valor de retorno debe ser -1, 0 ó 1, dependiendo de si el primer ítem precede al segundo en el orden establecido, si son iguales o si el segundo precede al primero, respectivamente.

Por ejemplo, si para nuestra aplicación establecemos que el orden depende del valor del dato del ítem, de menor a mayor, devolveremos -1 si *itemData1* es menor que *itemData2*, 0 si son iguales y 1 si el valor de *itemData1* es mayor que *itemData2*.

O como en el ejemplo 74, hemos definido un orden para mostrar las comidas según su tipo, primero las comidas, después los postres y al final las bebidas. Dentro de cada tipo se aplica el orden alfabético:

```
case WM_COMPAREITEM:
    lpcis = (LPCOMPAREITEMSTRUCT) lParam;
    /* Establecer un orden:
       1) Por tipos, primero comidas, después postres
       y último bebidas
       2) Dentro de cada tipo, usar el orden
       alfabético */
    if(comida[lpcis->itemData1].tipo == comida[lpcis->itemData2].tipo)
        return strcmp(comida[lpcis->itemData1].nombre,
            comida[lpcis->itemData2].nombre);
    else if(comida[lpcis->itemData1].tipo == 'c')
        return -1;
    else if(comida[lpcis->itemData1].tipo == 'b')
        return 1;
    else if(comida[lpcis->itemData2].tipo == 'c')
        return 1;
    else return -1;
    break;
```

## Ejemplo 74

# Capítulo 44 Control scrollbar avanzado

Terminamos el repaso de los controles más comunes del API de Windows hablando del control de barra de desplazamiento. Vamos a completar el contenido del [capítulo 12](#) con las explicaciones de todos los mensajes y características que no se mencionaron entonces.

## Controles de barra de desplazamiento y barras estándar

Las barras de desplazamiento estándar son las que suelen aparecer en la zona derecha e inferior de algunas ventanas.

Aunque aparentemente se trata de cosas similares, existen algunas diferencias entre los controles de barra de desplazamiento y las barras de desplazamiento estándar.

Para empezar, las barras estándar están siempre fuera del área de cliente, mientras que los controles son ventanas que están situadas dentro del área de cliente.

Otra diferencia es que las barras estándar están diseñadas para permitir desplazar el contenido del área de cliente, y para crearlas basta con indicar los estilos de ventana [WS\\_HSCROLL](#), [WS\\_VSCROLL](#) o ambos. Los controles de barra de desplazamiento se crean como el resto de los controles, ya sea mediante un fichero de recursos o bien insertados durante la ejecución, mediante las funciones [CreateWindow](#) o [CreateWindowEx](#).

Otra diferencia es que los controles de barra de desplazamiento, como ventanas que son, pueden recibir el foco del teclado, cosa que no sucede con las barras estándar.

Los controles de barra de desplazamiento también tienen un interfaz de teclado interno.

## Insertar controles scrollbar durante la ejecución

Evidentemente, también es posible insertar controles de barra de desplazamiento durante la ejecución. En este caso tendremos que insertar una ventana de la clase "SCROLLBAR". Para insertar el control también usaremos las funciones [CreateWindow](#) y [CreateWindowEx](#).

```
    HWND hctrl;
    ...
    case WM_CREATE:
        hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
        /* Insertar control Edit */
        hctrl = CreateWindowEx(
            0,
            "SCROLLBAR",      /* Nombre de la clase */
            NULL,             /* Texto del título */
            SBS_HORZ |
            WS_CHILD | WS_VISIBLE | WS_TABSTOP, /* Estilo
*/
            5, 5,             /* Posición */
            120, 85           /* Tamaño */
            hwnd,             /* Ventana padre */
            (HMENU)ID_SCROLL, /* Identificador del control
*/
            hInstance,        /* Instancia */
            NULL);            /* Sin datos de creación de
ventana */
```

Como vemos, usamos los mismos valores que en el fichero de recursos: un identificador, la clase de ventana (en este caso "SCROLLBAR"), una combinación de estilos, la posición y las dimensiones.

Al igual que en los demás controles, el identificador del control se suministra a través del parámetro *hMenu*, por lo que será necesario hacer un casting a [HMENU](#).

## Cambiar colores

No tenemos demasiado control sobre los colores de los controles de barra de desplazamiento. Sólo podemos modificar el color de la zona sobre la que se desplaza el cursor de la barra.

Para ello debemos procesar el mensaje [WM\\_CTLCOLORSCROLLBAR](#). Como en otros casos, podemos cambiar el color del fondo, usando la función [SetBkColor](#), y debemos devolver un pincel del color usado para pintar ese fondo:

```
case WM_CTLCOLORSCROLLBAR:
    SetBkColor((HDC)wParam, RGB(0,255,0));
    return (LRESULT)pincel;
```

## Estilos de scrollbar

En el [capítulo 12](#) sólo mencionamos los estilos básicos [SBS\\_HORZ](#) y [SBS\\_VERT](#), pero existen algunos más que veremos a continuación.

### Estilos de orientación

Los controles de barra de desplazamiento pueden tener dos orientaciones: horizontal o vertical. El estilo [SBS\\_HORZ](#) sirve para crear un control de barra de desplazamiento horizontal. El estilo [SBS\\_VERT](#), uno vertical.

En el primer caso, si no se especifican los estilos [SBS\\_BOTTOMALIGN](#) o [SBS\\_TOPALIGN](#), la barra tendrá la altura,

anchura y posición especificados por los parámetros usados en la función [CreateWindow](#) o [CreateWindowEx](#).

En el segundo caso, si no se especifican los estilos [SBS\\_RIGHTALIGN](#) o [SBS\\_LEFTALIGN](#), la barra tendrá la altura, anchura y posición especificados por los parámetros usados en la función [CreateWindow](#) o [CreateWindowEx](#).

## Alineamiento con los bordes

Existen cuatro estilos para alinear el control con cada uno de los cuatro bordes del rectángulo indicado en la función [CreateWindow](#) o [CreateWindowEx](#):

[SBS\\_BOTTOMALIGN](#) para alinear un control vertical con el borde superior del rectángulo.

[SBS\\_TOPALIGN](#) para alinear un control vertical con el borde superior del rectángulo.

[SBS\\_LEFTALIGN](#) para alinear un control horizontal con el borde izquierdo del rectángulo.

[SBS\\_RIGHTALIGN](#) para alinear un control horizontal con el borde derecho del rectángulo.

Si se usa cualquiera de estos estilos, el control de barra de desplazamiento tendrá la anchura por defecto para las barras de desplazamiento del sistema, independientemente del tamaño del rectángulo usado en la función.

## Opciones para cajas de tamaño

Las cajas de tamaño son barras de desplazamiento con un aspecto diferente. Seguro que las conoces, son los controles que nos permiten cambiar el tamaño de una ventana, y que suelen encontrarse en la esquina inferior derecha de las ventanas que pueden cambiar de tamaño. A veces, estos controles son invisibles, y sólo se distinguen porque cambia el cursor del ratón, mostrando dos flechas con la orientación arriba-izquierda y abajo-derecha. En

otras ocasiones son visibles mediante un mapa de bits consistente en tres barras diagonales y paralelas.

Nosotros podemos usar este tipo de controles (aunque su utilidad es francamente limitada), mediante barras de desplazamiento con los estilos siguientes:

[SBS\\_SIZEBOX](#) crea un control de caja de tamaño. Si no se especifican ninguno de los estilos [SBS\\_SIZEBOXBOTTOMRIGHTALIGN](#) o [SBS\\_SIZEBOXTOPLEFTALIGN](#), el control tendrá la altura, anchura y posición especificados por los parámetros de la llamada a la función [CreateWindow](#) o [CreateWindowEx](#).

[SBS\\_SIZEGRIP](#) lo mismo, pero el control se muestra con un borde realzado (sólo funciona correctamente en Windows 95).

## Alineamiento de cajas de tamaño

[SBS\\_SIZEBOXBOTTOMRIGHTALIGN](#) alinea la esquina inferior derecha del control con la esquina inferior derecha del rectángulo definido por los parámetros de la función [CreateWindow](#) o [CreateWindowEx](#).

[SBS\\_SIZEBOXTOPLEFTALIGN](#) alinea la esquina superior izquierda del size box con la esquina superior izquierda del rectángulo definido por los parámetros de la función [CreateWindow](#) o [CreateWindowEx](#).

Si se usa cualquiera de estos dos estilos, la caja de tamaño tendrá el tamaño por defecto para las cajas de tamaño del sistema.

Estos estilos se deben usar conjuntamente con [SBS\\_SIZEBOX](#) o [SBS\\_SIZEGRIP](#).

## Mostrar u ocultar barras de desplazamiento

En ocasiones nos puede interesar inhibir u ocultar barras de desplazamiento. Esto es frecuente con las barras estándar, por ejemplo, cuando todo el contenido a mostrar en una ventana cabe

en el área de cliente, podemos optar por inhibir las barras de desplazamiento, de modo que sigan siendo visibles, aunque no respondan al usuario, o podemos optar por ocultarlas, dejando que el espacio que ocupan pueda ser usado por el área de cliente.

Con nuestros controles de barra de desplazamiento podemos hacer lo mismo, para ello disponemos de dos funciones:

La función `ShowScrollBar` nos permite mostrar u ocultar un control de barra de desplazamiento o una barra estándar.

Si se trata de una barra estándar, en el primer parámetro indicaremos la ventana a la que pertenece mediante su manipulador. En el segundo parámetro indicaremos qué barra o barras queremos ocultar o mostrar: `SB_HORZ` para la horizontal, `SB_VERT` para la vertical o `SB_BOTH` para ambas. En el tercer parámetro indicaremos el valor `TRUE` para hacer visible la barra o `FALSE` para ocultarla:

```
ShowScrollBar(hWnd, SB_VERT, FALSE); // ocultar barra
estándar vertical
ShowScrollBar(hWnd, SB_HORZ, TRUE);  // mostrar barra
estándar horizontal
```

Si se trata de un control de barra de desplazamiento, en el primer parámetro deberemos indicar el manipulador del control. En el segundo parámetro usaremos el valor `SB_CTL`. El tercero tiene el mismo significado que en el caso anterior:

```
ShowScrollBar(GetDlgItem(hWnd, ID_SCR1), SB_CTL, FALSE);
// ocultar control de barra de desplazamiento
```

## Deshabilitar o habilitar un control de barra de desplazamiento

Como con cualquier otro control, es posible habilitar o deshabilitar cualquier control de barra de desplazamiento mediante una llamada a la función [EnableWindow](#):

```
EnableWindow(GetDlgItem(hwnd, ID_SCR1), TRUE); //
Habilitar
EnableWindow(GetDlgItem(hwnd, ID_SCR1), FALSE); //
Deshabilitar
```

## Deshabilitar o habilitar flechas

Por último, también podemos habilitar o deshabilitar una o ambas flechas de un control de barra de desplazamiento o de una barra de desplazamiento estándar. Bueno, en realidad, deshabilitar ambas flechas equivale a deshabilitar el control completo, pero esto nos da dos formas diferentes de conseguir el mismo resultado.

### Usando funciones

Para habilitar o deshabilitar las flechas de una barra de desplazamiento se usa la función [EnableScrollBar](#).

Si se trata de una barra estándar, el primer parámetro debe contener el manipulador de la ventana a la que pertenece la barra. En caso de tratarse de un control, debe ser el manipulador de ventana del propio control.

Para barras estándar, el segundo parámetro puede ser uno de los valores: [SB\\_HORZ](#), [SB\\_VERT](#) o [SB\\_BOTH](#), para referirse a la barra estándar horizontal, vertical o ambas, respectivamente. En caso de un control, el valor debe ser [SB\\_CTL](#).

El tercer parámetro puede tener uno de los valores siguientes:

- [ESB\\_DISABLE\\_UP](#): para deshabilitar la flecha hacia arriba en una barra vertical.



- **ESB\_DISABLE\_LEFT**: para deshabilitar la flecha hacia la izquierda en una barra horizontal.
- **ESB\_DISABLE\_LTUP**: equivale a las dos anteriores.
- **ESB\_DISABLE\_DOWN**: para deshabilitar la flecha hacia abajo en una barra vertical.
- **ESB\_DISABLE\_RIGHT**: para deshabilitar la flecha hacia la derecha en una barra horizontal.
- **ESB\_DISABLE\_RTDN**: equivale a las dos anteriores.
- **ESB\_DISABLE\_BOTH**: para deshabilitar ambas flechas.
- **ESB\_ENABLE\_BOTH**: para habilitar ambas flechas.

```
EnableScrollBar(hwnd, SB_HORZ, ESB_DISABLE_RIGHT);
EnableScrollBar(GetDlgItem(hwnd, ID_SCR1), SB_CTL,
ESB_DISABLE_UP);
```

Ya hemos comentado que deshabilitar ambas flechas equivale a deshabilitar la barra completa. Esto es tanto así que para habilitar un control deshabilitado por completo se puede usar tanto la función **EnableScrollBar** con el valor **ESB\_ENABLE\_BOTH** como la función **EnableWindow** con el valor **TRUE**.

De forma simétrica, deshabilitar ambas flechas mediante la función **EnableScrollBar**, ya sea una después de otra o ambas a la vez, equivale a usar la función **EnableWindow** con el valor **FALSE**.

## Usando mensajes

Además de la función **EnableScrollBar**, también podemos habilitar o deshabilitar flechas de barras de desplazamiento usando el mensaje **SBM\_ENABLE\_ARROWS**.

En este caso, el parámetro **lParam** no se usa, y debe valer 0. En el parámetro **wParam** usaremos el mismo valor que en el tercer parámetro de la función **EnableScrollBar**.

Este mensaje sólo sirve para habilitar o deshabilitar flechas en un control de barra de desplazamiento, y no en una barra estándar.

Por supuesto, se debe enviar a una ventana, usando el manipulador de ventana del control.

Como siempre, podemos usar dos funciones para enviar mensajes a un control: [SendMessage](#) o [SendDlgItemMessage](#). La primera función la usaremos cuando conozcamos el manipulador de la ventana que debe recibir el mensaje, aunque podríamos usar la función [GetDlgItem](#) para obtener ese manipulador, cuando no lo tengamos será más sencillo usar la segunda función. De este modo, estas tres funciones son equivalentes:

```
HWND hctl = GetDlgItem(hwnd, ID_SCR1);
SendMessage(hctl, SBM_ENABLE_ARROWS, ESB_DISABLE_UP, 0);
SendMessage(GetDlgItem(hwnd, ID_SCR1), SBM_ENABLE_ARROWS,
ESB_DISABLE_UP, 0);
SendDlgItemMessage(hwnd, ID_SCR1, SBM_ENABLE_ARROWS,
ESB_DISABLE_UP, 0);
```

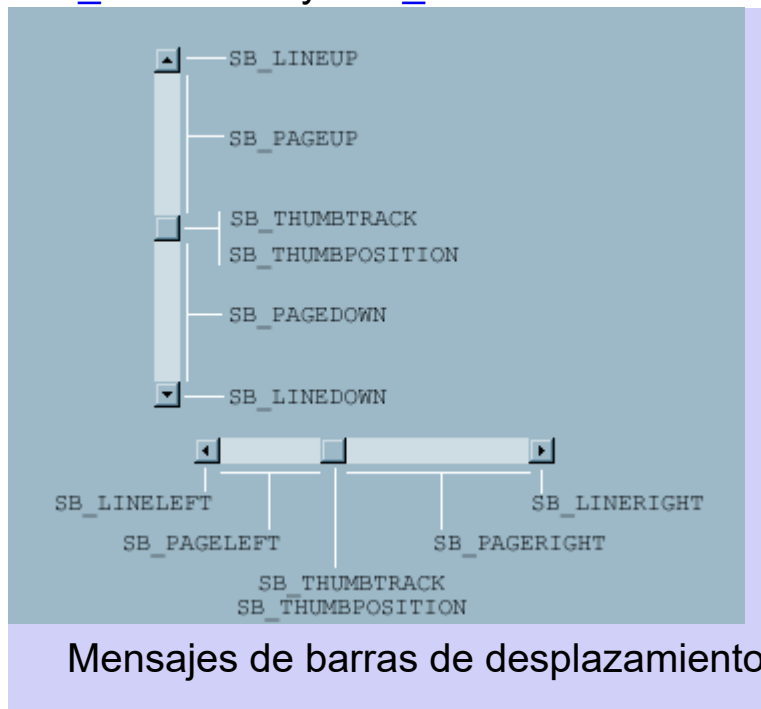
También hemos comentado con anterioridad que a pesar del nombre, la función [SendDlgItemMessage](#) sirve para enviar mensajes a controles que estén tanto en cuadros de diálogo como en ventanas normales.

## Mensajes de barras de desplazamiento

Los controles de barra de desplazamiento no envían mensajes de notificación. Al contrario que otros controles, todas las barras de desplazamiento envían mensajes [WM\\_HSCROLL](#) o [WM\\_VSCROLL](#), dependiendo de si se trata de barras horizontales o verticales, respectivamente.

Todo esto se comentó con suficiente detalle en el [capítulo 12](#), por lo que no abundaremos en este tema. Tan sólo añadir un gráfico con los códigos de notificación asociados a cada parte de la barra de desplazamiento. Recordemos que estos códigos se reciben en la

palabra de menor peso del parámetro wParam de los mensajes [WM\\_HSCROLL](#) y [WM\\_VSCROLL](#).



## Respuesta al teclado

Los controles de barra de desplazamiento tienen un interfaz de teclado predefinido, de modo que cuando tienen el foco, ciertas pulsaciones de teclas se convierten en mensajes

[WM\\_HSCROLL](#) o [WM\\_VSCROLL](#).

En la siguiente tabla vemos las teclas que se procesan, y qué códigos de notificación generan:

Tecla	Código de notificación	
	Vertical	Horizontal
Flecha arriba	SB_LINEUP	SB_LINELEFT
Flecha abajo	SB_LINEDOWN	SB_LINERIGHT
Flecha derecha	SB_LINEDOWN	SB_LINERIGHT
Flecha izquierda	SB_LINEUP	SB_LINELEFT
Página abajo	SB_PAGEDOWN	SB_PAGERIGHT
Página arriba	SB_PAGEUP	SB_PAGELLEFT

Fin	SB_BOTTOM
Inicio	SB_TOP

Vemos que en esta tabla aparecen dos códigos de notificación que no se comentaron en el punto anterior: **SB\_TOP** y **SB\_BOTTOM**. Esto se debe a que estos dos códigos sólo pueden ser generados mediante el teclado.

En un control de barra de desplazamiento vertical, **SB\_TOP** indica que el cursor se debe desplazar al tope superior, y **SB\_BOTTOM** al inferior. En controles horizontales indica los extremos izquierdo y derecho del control, respectivamente.

Las barras de desplazamiento estándar no tienen este interfaz de teclado definido, pero se puede hacer algo análogo procesando mensajes **WM\_KEYDOWN** y enviando correspondientes mensajes **WM\_HSCROLL** y **WM\_VSCROLL** a la ventana.

## Ejemplo 75

### Desplazar contenido de ventanas

El uso más común de las barras de desplazamiento estándar es permitir la posibilidad de mostrar distintas áreas de un documento cuando éste no puede ser mostrado íntegramente en el área de cliente.

Cuando se responde a códigos de notificación como **SB\_PAGEUP**, **SB\_PAGEDOWN**, **SB\_TOP**, etc, generalmente será necesario generar el contenido completo del área de cliente. Sin embargo, ante códigos como **SB\_LINEUP** o **SB\_LINERIGHT**, sólo una pequeña parte del área de cliente queda invalidada, el resto sigue siendo válida, después de desplazarla algunos pixels en la dirección adecuada.

Estas operaciones de desplazamientos de mapas de bits son tan frecuentes que, inevitablemente, el API dispone de funciones para realizarlas. Se trata de las funciones **ScrollWindow** y

[ScrollWindowEx](#), de las que es recomendable usar la segunda, ya que la primera se mantiene sólo por compatibilidad con versiones anteriores del API.

Esta función tiene bastantes parámetros, pero veremos que algunos de ellos no tendrán mucho uso en la mayor parte de los casos.

El primer parámetro es el manipulador de la ventana cuya área de cliente queremos desplazar.

El segundo indica las unidades lógicas a desplazar horizontalmente. Los valores negativos indican desplazamientos a la izquierda.

El tercer parámetro indica las unidades lógicas a desplazar verticalmente. Los valores negativos son desplazamientos hacia arriba.

El cuarto parámetro es un puntero a una estructura [RECT](#) que indica la zona rectangular, dentro del área de cliente, que queremos desplazar. Si usamos el valor [NULL](#) se tomará el rectángulo correspondiente al área de cliente completa.

El quinto parámetro también es un puntero a una estructura [RECT](#) que define un área rectangular de recorte. Sólo los puntos dentro de ese área se verán afectados, de modo que los puntos fuera de ella que después del desplazamiento estén dentro se pintarán, pero los que estén dentro que posteriormente queden fuera, no. De nuevo, si se especifica el valor [NULL](#), el rectángulo de recorte coincidirá con el del parámetro anterior.

El sexto parámetro es un manipulador de región. Antes de la llamada, esa región puede contener la zona de actualización actual, y la función la modificará para incluir las nuevas zonas que deben ser actualizadas. Si no nos interesa usar esta región, de nuevo podemos usar el valor [NULL](#).

El séptimo parámetro es un puntero a una estructura [RECT](#), la función actualiza ese rectángulo con la región rectangular invalidada después de realizado el desplazamiento. También podemos ignorar este dato usando el valor [NULL](#).

El octavo y último parámetro es una bandera que puede tomar tres valores diferentes:

- **SW\_INVALIDATE**: invalida la región identificada por el sexto parámetro una vez realizado el desplazamiento.
- **SW\_ERASE**: Borra la región invalidada enviando un mensaje **WM\_ERASEBKGD** a la ventana, cuando se especifica junto a valor **SW\_INVALIDATE**.
- **SW\_SCROLLCHILDREN**: desplaza también las ventanas hijas incluidas en el área a desplazar. Además, se envía un mensaje **WM\_MOVE** a cada una de las ventanas interseccionadas por ese área, aunque no hayan sido movidas.

```
ScrollWindowEx(hwnd, 10, 0, &re, NULL, NULL, NULL,  
SW_INVALIDATE);
```

El uso de esta función implica, generalmente, procesar el mensaje **WM\_PAINT**.

Otra función útil para desplazar el contenido de una ventana es **ScrollDC**. Esta función es muy parecida a la anterior, pero en lugar de un manipulador de ventana, se usa un manipulador de contexto de dispositivo, y no dispone del parámetro de banderas.

Existen además dos funciones que pueden resultar útiles para usar conjuntamente, como son **UpdateWindow** y **RedrawWindow**. La primera sirve para actualizar el área de cliente. Para ello se envía un mensaje **WM\_PAINT**, si la región de actualización no es nula, evitando la cola, con lo que se procesará inmediatamente, el envío del mensaje se omite si esta región está vacía.

La segunda función es algo más complicada, pero tiene un funcionamiento parecido. También sirve para actualizar una zona de la ventana, pero al contrario que la anterior, podemos especificar un rectángulo o una región de actualización.

El primer parámetro es un manipulador de la ventana que queremos actualizar.

El segundo es un puntero a una estructura [RECT](#) que contiene el rectángulo de actualización.

El tercero es un manipulador de una región de actualización, si se especifica una región en este parámetro, se ignora cualquier valor del anterior. Si ambos son [NULL](#), se actualiza toda el área de cliente.

El último parámetro son banderas que pueden afectar al modo en que trabaja la función, validando o invalidando la región especificada, indicando si debe actualizar o borrar el contenido, u opciones sobre las ventanas hijas. Ver la descripción de la función [RedrawWindow](#) para más detalles.

```
RedrawWindow(hwnd, NULL, NULL, RDW_UPDATENOW |  
RDW_ALLCHILDREN);
```

## Colores y medidas

Generalmente, usaremos los colores y medidas estándar para las barras de desplazamiento. Ya hemos visto que el único color que podemos personalizar es el del fondo de la zona de desplazamiento en la barra, el resto corresponde con los colores normales de los botones, ya que las flechas y el cursor se comportan, en cierto modo, como tales.

En cuanto a los colores, ya sabemos que podemos usar las funciones [GetSysColor](#) y [SetSysColors](#) para obtener o modificar, respectivamente, los valores de color del sistema. Concretamente, para este caso, nos interesa el valor de [COLOR\\_SCROLLBAR](#), que es el color del fondo de la zona de desplazamiento.

## Valores de medidas del sistema

Las barras de desplazamiento estándar horizontales tienen definidas unas medidas en el sistema: [SM\\_CXHSCROLL](#) para la

anchura y **SM\_CYHSCROLL** para la altura. Del mismo modo, las verticales tienen unas medidas **SM\_CXVSCROLL** para la anchura y **SM\_CYVSCROLL** para la altura.

Para obtener estos valores se puede usar la función **GetSystemMetrics**, indicando la constante que interese.

```
int anchura;  
  
anchura = GetSystemMetrics(SM_CXHSCROLL);
```

Los valores que podemos obtener mediante **GetSystemMetrics** son los siguientes:

Constante	Descripción
<b>SM_CXHSCROLL</b>	Anchura del mapa de bits de la flela en una barra de desplazamiento horizontal.
<b>SM_CXHTHUMB</b>	Anchura de la caja de desplazamiento en una barra horizontal. A partir de la versión 4.0, este valor recupera la anchura de una barra de desplazamiento cuyo tamaño de página es cero.
<b>SM_CXVSCROLL</b>	Anchura del



	mapa de bits de la flecha en una barra de desplazamiento vertical.
SM_CYHSCROLL	Altura del mapa de bits de la flecha en una barra de desplazamiento horizontal.
SM_CYVSCROLL	Altura del mapa de bits de la flecha en una barra de desplazamiento vertical.
SM_CYVTHUMB	Altura de la caja de desplazamiento en una barra vertical. A partir de la versión 4.0, este valor recupera la altura de una barra de desplazamiento cuyo tamaño de página es cero.

## Otros mensajes

Además de todos los mensajes comentados en el capítulo 12 y en este, hasta ahora, existen dos de los que no hemos hablado.

El primero de ellos es `SBM_GETRANGE`, que nos sirve para recuperar los valores mínimo y máximo del rango de un control de barra de desplazamiento.

En el parámetro `wParam` enviaremos un puntero a un entero, en ese entero recibiremos el valor mínimo del rango. En el parámetro `lParam` enviaremos otro puntero a un entero, en este recibiremos el valor máximo del rango.

El segundo mensaje es `SBM_SETRANGEREDRAW`, que se comporta de forma idéntica a `SBM_SETRANGE`, salvo que además de asignar un rango al control de barra de desplazamiento, también redibuja el control para mostrar su nuevo estado.

Como en el caso de `SBM_SETRANGE`, en el parámetro `wParam` pasaremos el valor mínimo del rango, y en `lParam`, el máximo.

## Ejemplo 76

# Capítulo 45 La impresora

Cuando, en el [capítulo 16](#), hablamos del GDI y de los Contextos de Dispositivo (DCs), comentamos que nuestras aplicaciones Windows no acceden directamente a los dispositivos de salida, sino que lo hacen a través de un interfaz, el DC, de modo que todo lo comentado entre los capítulos 16 y 29 es indiferente de si la salida es una pantalla o una impresora.

Pero esto es en lo que respecta al modo de enviar datos a un dispositivo. Cada impresora, como cada tarjeta gráfica, tiene sus particularidades: resolución, colores, orientación de papel, etc. Además, existen otras funcionalidades que podemos querer añadir a nuestras aplicaciones, como las vistas previas o la selección de páginas a imprimir o el número de copias.

## Proceso de impresión

Desde el punto de vista de la aplicación, no existe diferencia entre enviar datos a un DC de pantalla o de impresora. Sin embargo, para el sistema si existen diferencias, ya que la impresora no muestra los gráficos del mismo modo que una pantalla: es más lenta, no permite borrar, no permite superponer unas salidas gráficas a otras, o usan diferentes protocolos (por ejemplo, PostScript).

Por todo ello, normalmente no se envían los datos a la impresora hasta que la aplicación da por finalizada la creación de una página o incluso de un documento completo. Esto hace que deban existir varios procesos intermedios:

## El spooler de impresión (print spooler)

Un *spooler* no es otra cosa que un programa que gestiona un almacén temporal (un *buffer*) que puede estar almacenado en memoria o en disco duro, donde se almacena la información generada por un proceso o dispositivo y de donde se extrae por otro proceso o dispositivo, generalmente más lento.

En el caso del spooler de impresión, se almacenan los datos necesarios para generar el documento que se quiere imprimir a la espera de que la impresora, mucho más lenta, los procese y obtenga una copia impresa.

El spooler permite también crear colas de impresión. De hecho, esas colas son una necesidad, puesto que el ordenador puede generar documentos a una velocidad mucho mayor de la que la impresora puede imprimirlos, es imprescindible crear un proceso que impida que se mezclen páginas de unos documentos con los de otros, o incluso datos de distintas páginas en la misma.

También hace posible que la aplicación no deje de funcionar porque la impresora no esté conectada o esté temporalmente indisponible, por falta de papel, de tinta o de toner, por ejemplo. Los documentos se almacenan en la cola y se procesarán cuando la impresora vuelva a estar preparada.

Del mismo modo, permite seleccionar la impresora a la que se envía cada documento. Actualmente, en nuestros ordenadores puede haber conectadas varias impresoras: impresoras locales conectadas a distintos puertos, impresoras de red, impresoras remotas, impresoras virtuales para generar ficheros PDF o ficheros que se pueden enviar por correo o copiar e imprimir en otros lugares, faxes, etc.

## **El procesador de impresión (print processor)**

El procesador de impresión de Windows es una DLL que convierte los registros de un documento almacenado por el spooler a llamadas DDI (interfaz de controlador de dispositivo).

## La máquina de gráficos (graphics engine)

Es otra DLL que convierte la salida del procesador a llamadas a funciones de controlador de dispositivo. Este, a su vez, procesa esas llamadas y las convierte en comandos que la impresora puede manejar.

## El monitor

Una vez procesador todo el documento, el fichero de comandos de impresora se devuelve al spooler. El spooler es el encargado de enviar esos comandos a un monitor, otra DLL, que envía esos comandos a través del canal adecuado: red, puerto paralelo, serie... al dispositivo.

Todo este proceso es interno, y transparente para nosotros. Nuestra tarea se limita a seleccionar una impresora (o usar la impresora por defecto), y enviarle los documentos que queremos imprimir. El resto es tarea del sistema operativo.

## Obtener una lista de impresoras

Aunque en la práctica generalmente usaremos un cuadro de diálogo común (definido por el sistema) para seleccionar tanto la impresora como muchos otros parámetros de impresión: número de copias, rango de páginas, formato de página, etc.; a veces puede ser útil obtener una lista de las impresoras disponibles e información sobre cada una de ellas.

Para hacer esto disponemos de una función del API, [EnumPrinters](#) y de varias estructuras, dependiendo del tipo de información que precisemos sobre cada impresora.

Como en otras funciones de enumeración, la información de retorno se devuelve mediante un *array*, apuntado por uno de los parámetros. Pero no hay manera de saber el tamaño de ese *array*

hasta después de retornar, es decir, no tenemos a priori los parámetros necesarios para hacer la llamada.

Para solucionar esta situación [EnumPrinters](#) permite una llamada sin usar ese parámetro, y en otro de ellos retorna el tamaño necesario para el *array*. De este modo, mediante dos llamadas a la función podemos recuperar la información requerida:

```
    DWORD tamano, numero;
    PRINTER_INFO_4 *info;
...
    EnumPrinters(PRINTER_ENUM_LOCAL |
PRINTER_ENUM_CONNECTIONS,
                NULL,
                4,
                NULL,
                0,
                &tamano,
                &numero);
    info = (PRINTER_INFO_4*)malloc(tamano);
    EnumPrinters(PRINTER_ENUM_LOCAL |
PRINTER_ENUM_CONNECTIONS,
                NULL,
                4,
                (BYTE*)info,
                tamano,
                &tamano,
                &numero);
...
    free(info);
```

## Ejemplo 77

Para poder compilar este ejemplo hay que incluir entre las librerías "winspool.lib".

## Contexto de dispositivo

Ya hemos comentado que para enviar una salida a una impresora necesitamos un DC, pero los DC de impresora no son exactamente iguales que los de ventana que hemos usado hasta ahora. Para empezar, no podemos usar la función [GetDC](#), ya que los DC de impresora no están asociados a una ventana.

Para obtener un DC de impresora podemos usar dos funciones: [CreateDC](#) o [PrintDlg](#).

No hay una forma mejor que la otra, se trata de dos opciones que tenemos a nuestra disposición. Tal vez parezca más complicado usar [PrintDlg](#), pero eso sólo es porque ofrece muchas más opciones.

Típicamente, usaremos la primera forma para imprimir una copia completa de un documento. Es la respuesta a la opción "Imprimir" de los menús de las aplicaciones. La segunda opción la usaremos cuando queramos dejar al usuario seleccionar ciertas opciones: páginas a imprimir, número de copias, intercalado, impresora, etc.

## Usando CreateDC

Bien, la forma artesanal consiste en conseguir una lista de impresoras para que el usuario elija una, o usar la impresora por defecto. Crear un DC para la impresora, imprimir el documento, y destruir el DC:

```
char impresora[256];
int i;
DOCINFO di;
HBITMAP hBitmap;
HDC memDC;
BITMAP bm;

...
hBitmap = (HBITMAP)LoadImage(NULL, "meninas24.bmp",
IMAGE_BITMAP,
0, 0, LR_LOADFROMFILE);
GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&bm);
i = SendDlgItemMessage(hwnd, ID_LISTA, LB_GETCURSEL, 0,
0);
```

```

    SendDlgItemMessage(hwnd, ID_LISTA, LB_GETTEXT,
(WPARAM)i, (LPARAM)impresora);
    hPrinterDC = CreateDC("WINSPOOL", impresora, NULL,
NULL);
    di.cbSize = sizeof(DOCINFO);
    di.lpszDocName = "Prueba";
    di.lpszOutput = NULL;
    di.lpszDatatype = NULL;
    di.fwType = 0;
    StartDoc(hPrinterDC, &di);
    StartPage(hPrinterDC);
    TextOut(hPrinterDC, 10, 10, "Hola, mundo!", 12);
    Ellipse(hPrinterDC, 150, 180, 450, 670);
    EndPage(hPrinterDC);
    StartPage(hPrinterDC);
    memDC = CreateCompatibleDC(hPrinterDC);
    SelectObject(memDC, hBitmap);
    BitBlt(hPrinterDC, 250, 450, bm.bmWidth, bm.bmHeight,
memDC, 0, 0, SRCCOPY);
    DeleteDC(memDC);
    EndPage(hPrinterDC);
    EndDoc(hPrinterDC);
    DeleteDC(hPrinterDC);
    DeleteObject(hBitmap);

```

## Ejemplo 78

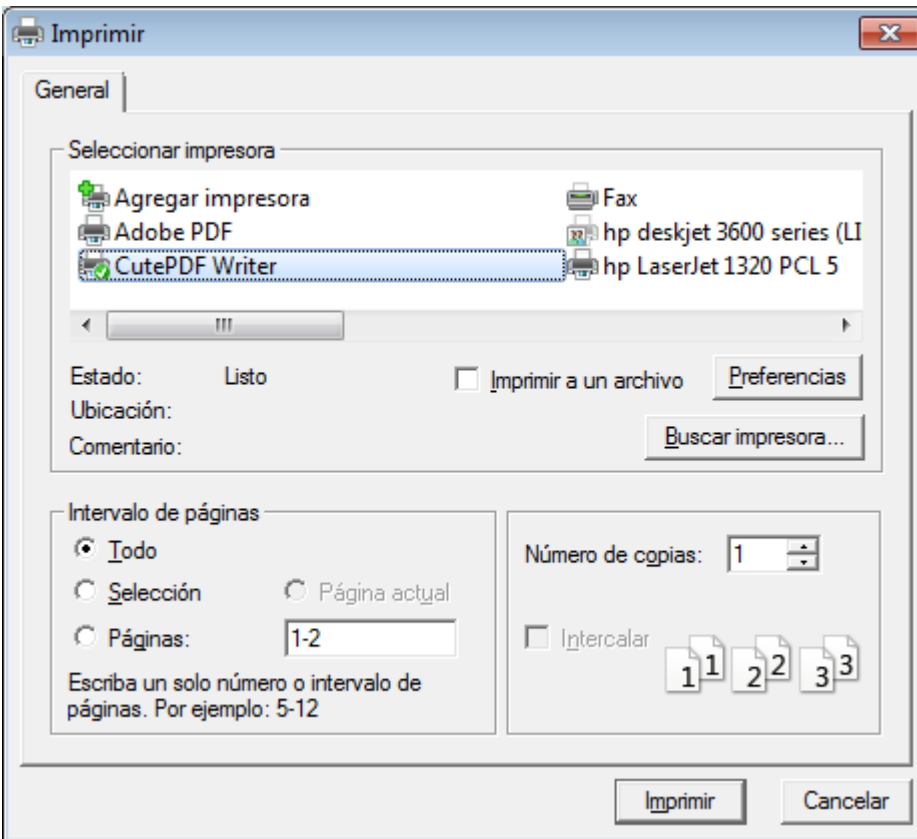
Para poder compilar este ejemplo hay que incluir entre las librerías "winspool.lib".

## Usando PrintDlg

Windows dispone de un cuadro de diálogo común para imprimir documentos. Usando ese cuadro podemos obtener del usuario todos los parámetros necesarios para configurar la impresión: impresora, páginas a imprimir, número de copias e intercalado de copias.

Para usarlo debemos invocar a la función [PrintDlg](#) usando como parámetro un puntero a una estructura [PRINTDLG](#). Previamente deberemos inicializar esa estructura con los datos adecuados para





### Diálogo de impresión

nuestro documento.

```
PRINTDLG pd
= {
sizeof(PRINTDLG),

hwnd, 0, 0,
0,
// hDC

PD_ALLPAGES
|
PD_RETURNDC
|
PD_USEDEVMO
DECOPIESAND
COLLATE,
1,
2, // Desde
la página 1
a la 2
```

```
1, 2, // La primera página es la 1, la última la 2
1, // Copias
NULL, 0, 0, 0, 0, 0, 0, 0};
```

Los valores que se suelen iniciar en la estructura son:

- El primer parámetro es el tamaño de la estructura **PRINTDLG**. Se debe indicar este tamaño ya que en distintas versiones del API la estructura puede haber añadido distintos campos.
- El segundo es un manipulador de la ventana padre del cuadro de diálogo.
- Los dos valores siguientes se usan para indicar la impresora. Si se inician a NULL (o 0), el cuadro de diálogo se encarga de inicializar esos valores con las impresora por defecto. Nosotros usaremos ceros. Al regresar, estos campos tienen manipuladores de memoria para objetos **DEVMODE** y **DEVNAMES** que nos indican la impresora seleccionada.

Nosotros deberemos liberar esa memoria cuando ya no la necesitemos. La estructura **DEVMODE** nos puede resultar útil para tener en cuenta las características de la impresora, dimensiones, si es color o blanco y negro, etc.

- El siguiente campo es un manipulador de DC. Si se usa el flag **PD\_RETURNDC**, al retornar este campo tendrá un DC para la impresora seleccionada. En realidad, esto es lo que necesitamos para imprimir. Igual que siempre, la aplicación debe liberar este DC cuando ya no lo necesite, mediante una llamada a **DeleteDC**.
- Las banderas, para indicar que queremos que se devuelva un DC para la impresora (**PD\_RETURNDC**), que inicialmente se seleccionan todas las páginas (**PD\_ALLPAGES**), y el valor **PD\_USEDEVMODECOPIESANDCOLLATE**, que indica que la opción de múltiples copias estará inhibido si la impresora no soporta esa opción.
- Los dos valores siguientes indican el rango de páginas seleccionadas. Cuando el usuario active la opción de "selección" se imprimirá ese rango de páginas.
- Los dos siguientes indican el número de la primera y última paginas. Estos valores se usan para verificar si el rango introducido por el usuario es o no válido.
- El siguiente valor indica el número de copias, si la impresora seleccionada soporta varias copias, si no, se ignora. Al retornar indica el número de copias pedidas por el usuario.
- El resto de los valores no nos interesan de momento, ya que usaremos el recurso por defecto.

```
case CM_IMPRIMIR:
    if(PrintDlg(&pd)) {
        ImprimirDocumento(&pd);
        GlobalFree(pd.hDevMode);
        GlobalFree(pd.hDevNames);
        DeleteDC(pd.hDC);
    }
    break;
```

Es importante liberar el manipulador del DC creado por `PrintDlg` y los manipuladores de memoria asignados para las estructuras `DEVNAMES` y `DEVMODE`.

## **Ejemplo 79**

# Capítulo 46 Controles comunes

Los controles que hemos visto hasta ahora son los básicos, en los próximos capítulos veremos el resto de los controles comunes añadidos mediante una DLL. Estos controles son:

- Control animation
- Listas de imágenes
- Barra de estado
- Barra de progreso
- Herramienta de sugerencia (Tooltip)
- Control Arriba-abajo (Up-Down)
- Control cabecera (Header)
- ComboBoxEx
- Selección de fecha y hora
- Calendario mensual
- Barra de guía (Trackbar)
- *Hot Key*
- *Barra de desplazamiento plana*
- *Control de dirección IP*
- *List Box mejorada, draglist*
- *List View*
- *Paginador*
- *Hojas de propiedades (Property Sheet)*
- *Rebar*
- *Editor de texto enriquecido (Rich Edit)*
- *SysLink*
- *Pestañas*
- *Barras de herramientas*
- *Tree View*
- *Task Dialog \**

Todos estos controles se han ido añadiendo al API a medida que aparecían nuevas versiones de Windows o de Internet Explorer. Nos facilitan mucho las cosas a la hora de obtener del usuario valores dentro de dominios específicos (fechas, texto enriquecido, teclas, direcciones IP...), o para crear barras de herramientas, barras de estado, o para añadir funcionalidades a los cuadros de diálogo, como animaciones, imágenes, barras de progreso, etc.

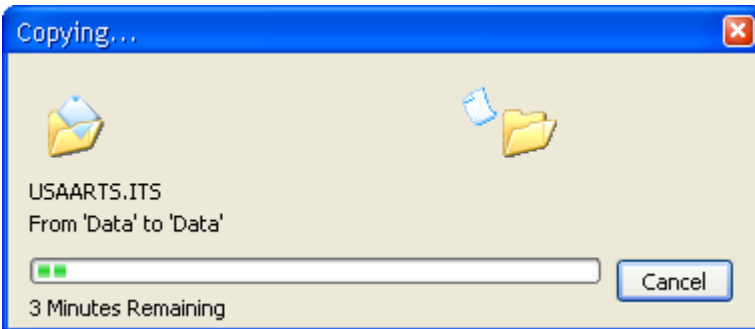
Empezaremos por estos últimos, ya que algunos de ellos los necesitaremos para sacar todo su rendimiento al resto.

¿Hay algo nuevo en los controles básicos?

Veremos que sí, y daremos un repaso al final. Los estilos visuales disponibles desde Windows XP afectan a aspecto gráfico de todos los controles, pero puede que haya más novedades...

- Button
- ComboBox
- Edit Control
- Scroll Bar
- Static Control

# Capítulo 47 Control animación



Diálogo copia ficheros

El control animación es un simple control estático que permite mostrar animaciones en formato AVI, sin sonido. Esto es importante, estos controles rechazarán

cualquier AVI que contenga sonido.

Seguramente has visto estos controles, por ejemplo al copiar, mover o borrar ficheros desde el administrador de archivos de Windows. Se suelen usar para indicar que se está realizando una tarea, en lugar de mostrar un cuadro de diálogo estático. De este modo, el usuario tiene la sensación de que su petición está siendo atendida y no que ha quedado atascada, sobre todo cuando lleva algún tiempo completarla.

En realidad sólo sirven como adorno, no aportan mejoras de rendimiento, ni tienen correspondencia con los procesos que se están ejecutando en el programa. Pero los programas tienen mejor aspecto (siempre que las animaciones tengan cierta calidad), y se ven más profesionales.

Para poder usar los controles animación en nuestros programas es necesario incluir el fichero de cabecera "commctrl.h", enlazar con la librería "comctl32" y asegurarse de que la dll ha sido cargada invocando a la función [InitCommonControlsEx](#).

**Nota:**

A menudo podemos olvidar invocar a `InitCommonControlsEx`, y a pesar de ello, el programa funciona correctamente. Esto se debe a que cualquier otro programa que se esté ejecutando actualmente (por ejemplo el propio IDE del compilador), ya ha cargado la dll en memoria. Si no tenemos esto en cuenta es probable que nuestro ejecutable no funcione correctamente cuando se ejecute en solitario.

La función `InitCommonControlsEx` requiere un parámetro de tipo `INITCOMMONCONTROLSEX`, en la que debemos asignar los dos miembros.

El miembro `dwSize` debe tomar el valor del tamaño de la estructura en bytes. Y en el miembro `dwICC` se debe especificar una combinación de banderas que indican que controles comunes queremos usar. En este caso, para el control de animación, el valor de la bandera es `ICC_ANIMATE_CLASS`:

```
INITCOMMONCONTROLSEX icCE;  
...  
icCE.dwSize = sizeof(INITCOMMONCONTROLSEX);  
icCE.dwICC = ICC_ANIMATE_CLASS;  
InitCommonControlsEx(&icCE);
```

## Ficheros de recursos

Como con los otros recursos que hemos visto con anterioridad, podemos integrar los recursos de animaciones dentro de un fichero de recursos.

En este caso tenemos dos recursos relacionados con los controles de animación. Uno es el propio recurso `AVI`, que contiene una animación en formato AVI sin sonido. El otro es el control que es una ventana de la clase "ANIMATE\_CLASS".

Para poder incluir este control en nuestros cuadros de diálogo hay que incluir el fichero de cabecera "commctrl.h".

```
#include <windows.h>
#include <commctrl.h>

mundo AVI "Globe.avi"
reloj AVI "L_Hourglass.avi"

LANGUAGE LANG_NEUTRAL, SUBLANG_NEUTRAL
"Dialogo" DIALOG 0, 0, 139, 50
STYLE DS_MODALFRAME | WS_CAPTION | WS_VISIBLE | WS_POPUP
CAPTION "Dialogo animación"
FONT 8, "Ms Shell Dlg"
{
    CONTROL        "reloj", -1, ANIMATE_CLASS, ACS_CENTER |
ACS_TRANSPARENT | ACS_AUTOPLAY, 22, 12, 20, 20
    DEFPUSHBUTTON  "Cerrar", IDOK, 79, 16, 50, 14
}
```

En este ejemplo de fichero de recursos hemos definido dos recursos de tipo [AVI](#), que se insertan desde un fichero externo.

Además, hemos definido un cuadro de diálogo en el que hemos insertado un control de la clase "ANIMATE\_CLASS". En el texto se indica el recurso [AVI](#) que se mostrará en el control. También se puede usar un identificador entero, si el recurso [AVI](#) tiene un identificador entero en lugar de uno de cadena.

Hemos indicado además tres estilos propios de los controles de animación:

- **ACS\_CENTER**: que centrará la animación en el control.
- **ACS\_TRANSPARENT**: tomará el color del fondo de la animación AVI como transparente.
- **ACS\_AUTOPLAY**: reproducirá la animación tan pronto como se cree el control.

Estos son, prácticamente, todos los estilos disponibles para estos controles. Existe otro, **ACS\_TIMER**, que indica que no se



debe crear un hilo para reproducir la animación en segundo plano. Pero este es el estilo por defecto desde la aparición de Windows XP.

Para más detalles sobre los estilos de controles de animación ver [estilos de animación](#).

## Insertar durante la ejecución

Podemos insertar controles de animación durante la ejecución del programa, directamente en nuestras ventanas.

El proceso es algo diferente al que hemos visto con otros controles. Primero, usaremos la macro [Animate\\_Create](#). Esta macro crea el control de animación, pero no tiene ningún parámetro que indique su posición o su tamaño. Tan sólo hay que suministrar un manipulador de su ventana padre, un identificador, los estilos y un manipulador de la instancia que crea el control.

Esta función invoca internamente a [CreateWindow](#).

A continuación moveremos el control a la posición deseada, usando la función [SetWindowPos](#). Esta función cambia la posición y tamaño de una ventana, pero también cambia su posición relativa en la lista de ventanas, lo que se conoce como orden-z. Ese orden, cuando se trata de controles, es en el que se activa cada uno de ellos cuando se pulsa la tecla TAB. Para establecer la posición del control en ese orden se usa el segundo parámetro, que indica el manipulador de la ventana anterior según ese orden. Además disponemos de algunas banderas.

Una vez colocado el control, abrimos una animación, usando la macro [Animate\\_Open](#). Puede ser una animación procedente de un recurso, identificada por su cadena identificadora o un identificador entero, creado mediante la macro [MAKEINTRESOURCE](#). En ese caso, se cargará el recurso desde el módulo especificado por el manipulador de instancia indicado en la llamada a [Animate\\_Create](#).

También puede ser un fichero .avi externo, que se cargará desde esta macro. En ese caso hay que especificar el nombre completo, y el camino, si no está en la misma carpeta que el fichero ejecutable.

Por último, mostramos el control usando la función [ShowWindow](#), indicando el modo [SW\\_SHOW](#).

```
        hctrl = Animate_Create(hwnd, IDC_ANIMATE,  
WS_BORDER | WS_CHILD | ACS_TRANSPARENT | ACS_AUTOPLAY,  
hInstance);  
        SetWindowPos(hctrl, 0, 10, 40, 48, 45,  
SWP_NOZORDER | SWP_DRAWFRAME);  
        Animate_Open(hctrl, "mundo");  
        ShowWindow(hctrl, SW_SHOW);
```

Cuando se destruya la ventana padre del control de animación deberemos cerrar esa animación, de modo que se borre de la memoria y se libere el recurso. Para ello usaremos la macro [Animate\\_Close](#).

```
        Animate_Close(GetDlgItem(hwnd, IDC_ANIMATE));
```

## Manipular la animación

El control de animación reproduce una animación, evidentemente. Pero esa animación puede estar inicialmente parada, si no se indica el estilo [ACS\\_AUTOPLAY](#), de modo que tendremos que reproducirla si queremos que no esté parada, o podemos pararla, o mostrar uno de sus fotogramas.

Para estas acciones disponemos de un juego de mensajes o macros (que lo único que hacen es enviar esos mensajes).

### Abrir animación

Cuando se crea un control de animación no se le asigna una animación, este proceso hay que realizarlo de forma explícita

mediante un mensaje [ACM\\_OPEN](#) o las macros [Animate\\_Open](#) o [Animate\\_OpenEx](#).

La macro [Animate\\_Open](#) abre un fichero externo en formato avi o un recurso, identificado por una cadena o por un entero. En caso de ser un entero se debe usar la macro [MAKEINTRESOURCE](#)(id, 0).

El primer parámetro es un manipulador de ventana del control. Una vez abierta la animación, se muestra el primer fotograma en el control.

La macro [Animate\\_OpenEx](#) es similar, pero se añade un parámetro entre el manipulador de ventana y el identificador del recurso, que identifica el módulo desde el que se cargará el recurso, mediante un manipulador de instancia.

Alternativamente, podemos usar el mensaje [ACM\\_OPEN](#), indicando en wParam el manipulador de instancia, o 0 si se quiere usar la instancia desde la que se creó el control, y en lParam el nombre o identificador del recurso o el nombre del fichero.

```
    Animate_Open(hctrl, "mundo");  
    Animate_Open(hctrl, "Gears.avi");  
    SendMessage(hctrl, ACM_OPEN, (WPARAM) 0,  
(LPARAM) MAKEINTRESOURCE(101, 0));
```

## Reproducir

Para reproducir una animación podemos usar el mensaje [ACM\\_PLAY](#) o la macro [Animate\\_Play](#).

Es más cómodo usar la macro, a la que deberemos pasar algunos parámetros. El primero, un manipulador del control animación en el que queramos reproducir la animación. El segundo es el número del primer fotograma que queramos reproducir, cero si es el primero. El tercer parámetro es el número del último parámetro a reproducir, -1 si es el último. El cuarto parámetro es el número de

veces que queremos que se repita la animación, o -1 si queremos que se repita indefinidamente.

Si optamos por el mensaje, los parámetros son los mismos. Enviaremos el mensaje al control de animación deseado. En wParam indicaremos el número de repeticiones o el valor -1 para repetir indefinidamente. En lParam empaquetaremos el fotograma de inicio y final usando la macro [MAKELONG](#)(inicio, final):

```
// Macro y mensaje equivalentes:  
Animate_Play(GetDlgItem(hwnd, IDC_ANIMATE), 0, -1,  
-1);  
SendMessage(GetDlgItem(hwnd, IDC_ANIMATE), ACM_PLAY,  
(WPARAM) (UINT) -1, (LPARAM) MAKELONG(0, -1));
```

## Detener

Para detener una animación usaremos el mensaje [ACM\\_STOP](#) o la macro [Animate\\_Stop](#).

Dado que para detener la animación no se necesitan parámetros, las dos formas son igualmente simples, y sólo necesitamos un manipulador de ventana del control:

```
// Macro y mensaje equivalentes:  
Animate_Stop(GetDlgItem(hwnd, IDC_ANIMATE));  
SendMessage(GetDlgItem(hwnd, IDC_ANIMATE), ACM_STOP,  
0, 0);
```

## Mostrar un fotograma

A veces podemos querer mostrar un único fotograma. Por ejemplo, podemos crear una animación con varios fotogramas que, aunque no se comporten como una película, cada fotograma indique el estado de determinadas condiciones. Para mostrar ese estado nos bastará con mostrar uno de los fotogramas, y nunca

necesitaremos reproducir la animación completa. Podríamos hacer lo mismo con varios mapas de bits o con varios iconos, por supuesto.

Para colocar la animación en un fotograma concreto podemos usar la misma macro o mensaje que para reproducirla, pero indicando el mismo fotograma de inicio y final. También disponemos de una macro específica para ello, [Animate\\_Sseek](#), que requiere dos parámetros: un manipulador del control animación, y el número del fotograma. Hay que tener en cuenta que los fotogramas empiezan a numerarse en 0, por lo tanto el tercer fotograma, por ejemplo, sería el número 2. Por ejemplo:

```
// Segundo fotograma:
Animate_Sseek(GetDlgItem(hwnd, IDC_ANIMATE), 1);
// Sexto fotograma:
SendMessage(GetDlgItem(hwnd, IDC_ANIMATE), ACM_PLAY,
(WPARAM) (UINT) 0, (LPARAM) MAKELONG(5, 5));
```

## Verificar reproducción

Según la documentación de Microsoft, podemos comprobar si una animación se está reproduciendo en un control animación usando el mensaje [ACM\\_ISPLAYING](#) o la macro [Animate\\_IsPlaying](#). Sin embargo, en la versión del API de la que dispongo actualmente, esta opción no está disponible.

En su lugar, podemos usar los mensajes de notificación, que veremos más abajo.

## Cerrar animación

Cada animación asignada a un control debe ser liberada o cerrada para liberar recursos o cerrar el fichero que la contiene. Esto se hace mediante la macro [Animate\\_Close](#) o el mensaje [ACM\\_OPEN](#).

La macro sólo necesita como parámetro un manipulador de ventana del control animación.

El mensaje es el mismo que para abrir la animación, pero usando el valor 0 como identificador de recurso.

```
    Animate_Close(hctrl);  
    SendDlgItemMessage(hwnd, IDC_ANIMATE2, ACM_OPEN, 0,  
0);
```

## Mensajes de notificación

Existen dos mensajes de notificación para los controles de animación. El mensaje de notificación **ACN\_START** se envía a la ventana padre del control cuando la animación empieza a reproducirse. El mensaje **ACN\_STOP** se envía cuando la animación se detiene.

Estos mensajes de notificación se reciben en el parámetro wParam de un mensaje **WM\_COMMAND**. En la palabra de menor peso se envía el identificador del control, en la de mayor peso el mensaje de notificación.

En este ejemplo usamos los mensajes de notificación para mostrar un mensaje en un control estático y para activar o desactivar los botones de marcha y paro de la animación:

```
    case WM_COMMAND:  
        switch(LOWORD(wParam)) {  
            case IDC_ANIMATE:  
                switch(HIWORD(wParam)) {  
                    case ACN_START:  
                        SetWindowText(GetDlgItem(hwnd,  
IDC_MENSAJE), "Marcha");  
                        EnableWindow(GetDlgItem(hwnd,  
CM_PLAY), FALSE);  
                        EnableWindow(GetDlgItem(hwnd,  
CM_STOP), TRUE);
```

```
                break;
            case ACN_STOP:
                SetWindowText(GetDlgItem(hwnd,
IDC_MENSAJE), "Parado");
                EnableWindow(GetDlgItem(hwnd,
CM_PLAY), TRUE);
                EnableWindow(GetDlgItem(hwnd,
CM_STOP), FALSE);
                break;
        }
        break;
    ...
}
```

## Ejemplo 80

# Capítulo 48 Listas de imágenes

Haremos un interludio en este capítulo. Las listas de imágenes no son estrictamente controles, al menos según mi opinión, aunque la documentación de Microsoft los considera como tales. Pero sí los usaremos en otros controles que veremos en próximos capítulos, como en las barras de herramientas, los TreeViews o los ListViews, por lo que será bueno conocerlos antes.

Las listas de imágenes se usan también para operaciones de arrastrar y soltar.

Para poder usar las listas de imágenes en nuestros programas es necesario incluir el fichero de cabecera "commctrl.h".

Una lista de imágenes es una colección de imágenes del mismo tamaño, a las que podemos acceder mediante un índice. Todas las imágenes se almacenan en un único mapa de bits, una a continuación de otra. Existe la opción de añadir un segundo mapa de bits monocromo con máscaras, de modo que las imágenes se puedan mostrar con una zona transparente, igual que los iconos.

El API suministra funciones y macros para crear y destruir listas de imágenes, añadir o eliminar imágenes de una lista, mostrarlas en pantalla, extraer iconos, etc.

## Crear una lista de imágenes

Consideraremos las listas de imágenes como un recurso. Y como tal, las crearemos cuando las necesitemos y las destruiremos cuando ya no sean necesarias. Generalmente usaremos los mensajes de creación de ventanas o diálogos para crear las listas de imágenes, y los mensajes de destrucción para destruirlas.

Crear una lista de imágenes requiere varios pasos. El primero es crear la propia lista, usando la función [ImageList\\_Create](#). Esta



función necesita varios parámetros:

- La anchura y altura de las imágenes que contiene.
- Unas banderas que indican la codificación de color y si se trata de una lista de imágenes con o sin máscara.
- El número de imágenes iniciales de la lista.
- Y el número de imágenes en el que la lista crecerá si es necesario añadir imágenes a ella.

```
static HIMAGELIST hIml;  
static HIMAGELIST hIml2;  
...  
    hIml = ImageList_Create(32, 32, ILC_COLOR24, 10, 4);  
    hIml2 = ImageList_Create(24, 24, ILC_COLOR8 |  
ILC_MASK, 15, 2);  
...
```

Este ejemplo crea dos listas de imágenes. La primera con 10 imágenes de 32x32 pixels y 24 bits de profundidad de color. Si fuera necesario añadir imágenes, la lista crecerá en grupos de cuatro imágenes. La segunda con 15 imágenes de 24x24 pixels y 8 bits de profundidad de color, y que crecerá en grupos de dos imágenes.

Esta función crea una lista vacía, en el punto siguiente veremos cómo añadir imágenes.

Una vez terminado el trabajo con las listas de imágenes procederemos a destruirlas, de modo que se liberen los recursos usados, para ello usaremos la función [ImageList\\_Destroy](#):

```
ImageList_Destroy(hIml);  
ImageList_Destroy(hIml2);
```

## Añadir y eliminar imágenes

Disponemos de una batería de funciones para añadir, eliminar y sustituir imágenes en una lista de imágenes.

Las dos funciones principales para añadir imágenes a una lista son [ImageList\\_Add](#) y [ImageList\\_AddMasked](#).

A pesar de lo que pueda indicar el nombre, las dos funciones pueden insertar imágenes con máscara, aunque la forma de indicar esa máscara es diferente en cada caso.

[ImageList\\_Add](#) toma tres parámetros. El primero es un manipulador de la lista de imágenes a la que se añadirá la imagen. El segundo es un manipulador de mapa de bits con la imagen a añadir. El tercero es un manipulador de mapa de bits con la máscara para la imagen. Si este parámetro es 0, no se usará máscara.

```
hbm1 = LoadBitmap(hInstance, "mapabits");
ImageList_Add(hIml, hbm1, 0);
DeleteObject(hbm1);

...

hbm2 = LoadBitmap(hInstance, "docmask");
hbm1 = LoadBitmap(hInstance, "doc1");
ImageList_Add(hIml2, hbm1, hbm2);
DeleteObject(hbm1);
DeleteObject(hbm2);
```

[ImageList\\_AddMasked](#) también toma tres parámetros. El primero es un manipulador de la lista de imágenes a la que se añadirá la nueva imagen. El segundo es un manipulador de mapa de bits con la imagen a añadir. El tercero es un valor [COLORREF](#) con el color que se usará para generar la máscara. Cada pixel de ese color en la imagen insertada corresponde con un bit puesto a uno en la máscara.

```
hbm1 = LoadBitmap(hInstance, "doc1");
ImageList_AddMasked(hIml2, hbm1, RGB(255,0,128));
DeleteObject(hbm1);
```

La función [ImageList\\_Replace](#) nos permite sustituir una imagen de una lista de imágenes. Los parámetros son los mismos que para [ImageList\\_Add](#), salvo que se añade un parámetro entero después del manipulador de la lista, que indica el índice de la imagen a sustituir.

También podemos añadir imágenes desde iconos, usando la macro [ImageList\\_AddIcon](#) o la función [ImageList\\_ReplaceIcon](#). En el caso de la macro [ImageList\\_AddIcon](#), el primer parámetro es un manipulador de la lista de imágenes a la que añadiremos la imagen, y el segundo un manipulador de icono.

```
hicon = LoadIcon(hInstance, "tierra");  
ImageList_AddIcon(hIml2, hicon);
```

En el caso de la función [ImageList\\_ReplaceIcon](#) el primer argumento es el mismo, el segundo es un entero que indica el índice de la imagen a sustituir dentro de la lista, y el tercero es un manipulador de icono. Si el segundo parámetro es -1, el icono se añade al final de la lista. Eso es lo que hace la macro [ImageList\\_AddIcon](#).

Disponemos de una función similar para sustituir imágenes a partir de manipuladores de mapas de bits, en lugar de iconos, [ImageList\\_Replace](#). El primer argumento es un manipulador de lista de imágenes, el segundo el índice de la imagen a sustituir, el tercero el manipulador del mapa de bits de la imagen, y el cuarto el manipulador del mapa de bits de la máscara.

```
hicon = LoadIcon(hInstance, "casa");  
hbm2 = LoadBitmap(hInstance, "docmask");  
hbm = LoadBitmap(hInstance, "doc1");  
ImageList_Replace(hIml2, 9, hbm, hbm2);  
DeleteObject(hbm2);  
DeleteObject(hbm);  
ImageList_ReplaceIcon(hIml, 9, hicon);
```

Por último, también es posible eliminar imágenes de una lista mediante la función [ImageList\\_Remove](#). Esta función sólo necesita dos argumentos. El primero, un manipulador de la lista de imágenes, y el segundo, el índice de la imagen a eliminar.

```
ImageList_Remove(hIml2, 9);
```

## Crear listas con imágenes

La función [ImageList\\_LoadImage](#) nos permite crear una lista de imágenes y añadir imágenes al mismo tiempo. Estos dos ejemplos son equivalentes:

```
hIml = ImageList_LoadImage(hInstance, "mapabits",  
32, 4, CLR_NONE, IMAGE_BITMAP, LR_CREATEDIBSECTION);
```

Y:

```
hIml = ImageList_Create(32, 32, ILC_COLOR24, 10,  
4);  
hbm = LoadBitmap(hInstance, "mapabits");  
ImageList_Add(hIml, hbm, 0);  
DeleteObject(hbm);
```

La función [ImageList\\_LoadImage](#) necesita siete parámetros. El primero es un manipulador de la instancia desde donde se carga el mapa de bits. También puede ser cero, si se cargan mapas de bits OEM (mapas de bits, iconos o cursores del sistema).

El segundo parámetro es un identificador de recurso, un nombre de fichero o un entero que identifique un recurso de sistema.

El tercero indica la anchura en pixels de cada imagen. El número de imágenes se calcula a partir de las dimensiones del mapa de

bits.

El cuarto parámetro indica la cantidad de imágenes en que crecerá la lista si es necesario añadir imágenes.

El quinto es el color utilizado para crear la máscara, o CLR\_NONE para si no se quiere generar máscara.

El sexto indica el tipo de imagen a cargar, IMAGE\_BITMAP, IMAGE\_ICON o IMAGE\_CURSOR.

El séptimo parámetro es un conjunto de banderas que especifican el modo de cargar la imagen. En nuestro caso hemos especificado LR\_CREATEDIBSECTION para indicar que se preserve el número de bits por color de la imagen original. Hay banderas para indicar que se cargue la imagen desde un fichero, para que sean transparentes, etc.

Otra función similar es [ImageList\\_LoadBitmap](#), salvo que sólo dispone de los primeros cinco parámetros, por lo que no es posible cargar mapas de bits desde ficheros ni usar mapas de bits de tipo DIB.

```
hIml = ImageList_LoadBitmap(hInstance,  
"mapabits", 32, 4, CLR_NONE);
```

La última función de este grupo es [ImageList\\_Merge](#). Esta función crea una nueva lista de imágenes con una única imagen que es el resultado de mezclar dos imágenes procedentes de dos listas de imágenes. Las dos listas pueden ser la misma. El primer y segundo parámetro son el manipulador de la lista de imágenes e índice de la primera imagen, el tercero y cuarto el manipulador de la lista e índice de la segunda imagen. El quinto y sexto es un desplazamiento de la segunda imagen sobre la primera.

```
hIml3 = ImageList_Merge(hIml2, 2, hIml2, 10, 0,  
0);
```

## Obtener iconos

La función [ImageList\\_GetIcon](#) nos permite extraer un icono desde una lista de imágenes. El primer parámetro es el manipulador de una lista de imágenes, el segundo parámetro es el índice de la imagen. El tercero es una combinación de banderas análogas a las de la función [ImageList\\_Draw](#). El resultado es el manipulador de un icono extraído de la lista.

```
HICON hicon;  
...  
hicon = ImageList_GetIcon(himl, 2, ILD_TRANSPARENT);
```

## Mostrar imágenes

Todo lo anterior suele ser suficiente para muchas de las aplicaciones de las listas de imágenes. Por ejemplo, los controles `ListView` y `TreeView` se encargan de mostrar las imágenes para cada ítem, según su estado de forma automática. Sin embargo, también podemos mostrar imágenes de una lista de imágenes desde nuestros programas, para ello disponemos de un par de funciones.

La función [ImageList\\_Draw](#) permite trazar una imagen de una lista de imágenes. Esta función requiere seis parámetros. El primero, un manipulador de la lista de imágenes. El segundo, un índice de la imagen a mostrar. El tercero, un manipulador de contexto de dispositivo de destino. El cuarto y quinto, las coordenadas donde se mostrará la imagen. Y el sexto el estilo de trazado.

Hay que tener en cuenta que si la lista de imágenes ha sido creada con la bandera `ILC_COLORDB`, `ILC_COLOR24` o `ILC_COLOR32`, los estilos resaltados `ILD_BLEND25` e `ILD_BLEND50` usarán una trama de puntos, y no se podrá distinguir

entre ellos. El resto de las banderas de color usan una mezcla para hacer los resaltados, y en general, tendrán un aspecto diferente si se usa `ILD_BLEND25` o `ILD_BLEND50`.

Los estilos `ILD_BLEND25` e `ILD_BLEND50` se añaden a `ILD_NORMAL` y `ILD_TRANSPARENT` para indicar que el usuario ha seleccionado la imagen.

Además, el modo *normal* con imágenes de listas sin máscara es equivalente al modo *transparente*, en ninguno de los dos casos hay transparencias.

Cuando estos estilos se aplican a imágenes que provienen de listas con máscaras, la diferencia es más sutil. Si se muestra en el estilo *normal* la parte del fondo enmascarada se muestra con el color de fondo de la lista de imágenes. Si se muestra en el estilo *transparente* la parte del fondo enmascarada conserva el contenido original.

Por último, el estilo `ILD_MASK` muestra la máscara, si existe.

```
ImageList_Draw(hIm1, 3, hDC, 10, 40, ILD_BLEND25 |  
ILD_NORMAL);
```

La función [ImageList\\_DrawEx](#) es más versátil. Además de los parámetros usados con [ImageList\\_Draw](#), se deben especificar algunos más, lo que nos proporciona un control adicional sobre el aspecto de las imágenes.

Los cinco primeros parámetros son los mismos que para [ImageList\\_Draw](#). Los dos siguientes indican la anchura y altura de la imagen a mostrar, pueden ser cero, para indicar que se muestre la anchura y altura por defecto. El octavo parámetro es un color usado para pintar el fondo. Puede ser cualquier color [RGB](#), el valor `CLR_NONE`, para indicar que no hay color de fondo o `CLR_DEFAULT` para indicar que se use el color de fondo por defecto. El noveno parámetro es un color de usado para el primer plano, y también puede tomar los valores `CLR_NONE` o

CLR\_DEFAULT. Este color sólo se usa para los resaltados ILD\_BLEND25 y ILD\_BLEND50, y no se tiene en cuenta si no aparecen esas banderas. Si se usa el valor CLR\_DEFAULT, el resultado es el mismo que con la función [ImageList\\_Draw](#). El último parámetro es el mismo que el último de [ImageList\\_Draw](#), e indica las banderas de estilo.

```
ImageList_DrawEx(hIml, 3, hDC, 10, 40, 32, 32,  
GetSysColor(COLOR_MENU), RGB(255,0,0) , ILD_BLEND25 |  
ILD_NORMAL);
```

## El color de fondo

Para cada lista de imágenes podemos establecer un color de fondo, que se usará para rellenar el espacio no enmascarado cuando se tracen imágenes en el estilo *normal*. Este color no afecta a listas de imágenes sin máscaras. Para asignar el color de fondo se usa la función [ImageList\\_SetBkColor](#), indicando como parámetros el manipulador de la lista de imágenes y el color de fondo que se quiere establecer. Para recuperar el color de fondo de una lista de imágenes podemos usar la función [ImageList\\_GetBkColor](#).

```
ImageList_SetBkColor(hIml, GetSysColor(COLOR_MENU));
```

## Imágenes superpuestas



En cada lista de imágenes se pueden indicar hasta cuatro imágenes que se podrán usar posteriormente como imágenes superpuestas. El efecto es similar al que usa Windows para crear los iconos de acceso directo, cuando



añade una pequeña flecha en una de las esquinas al icono de la aplicación.

Estas imágenes se conocen en el API como "overlay masks". Se puede usar cualquier imagen de la lista como imagen superpuesta, tan sólo hay que especificar cuales de ellas lo pueden ser usando la función [ImageList\\_SetOverlayImage](#). Hay que especificar tres parámetros. El primero un manipulador de la lista de imágenes, el segundo el índice de la imagen, y el tercero el número de la imagen superpuesta, entre 1 y 4.

Para trazar una imagen superpuesta se usan las funciones [ImageList\\_Draw](#) o [ImageList\\_DrawEx](#), el índice de la imagen superpuesta se indica mediante la macro [INDEXTOOVLAYMASK](#) añadida a las banderas de estilo.

```
ImageList_SetOverlayImage(hIml2, 6, 1);  
...  
ImageList_Draw(hIml2, 2, hDC, 10, 10,  
ILD_TRANSPARENT | INDEXTOOVLAYMASK(1));
```

## Ejemplo 81

### Arrastre de imágenes

Las operaciones de arrastrar y soltar siguen siempre un patrón parecido. Empiezan con una pulsación de ratón sobre el objeto a arrastrar. Mientras se mantiene pulsado, el objeto seleccionado se mueve junto con el cursor del ratón. Cuando se suelta el botón del ratón, el objeto se suelta en el punto actual del cursor.

Generalmente usaremos el botón izquierdo del ratón, eso quiere decir que deberemos procesar tres mensajes: [WM\\_LBUTTONDOWN](#), [WM\\_MOUSEMOVE](#) y [WM\\_LBUTTONUP](#).

#### Inicio del arrastre

Cuando detectemos la pulsación del botón del ratón, comprobaremos si el cursor está sobre el objeto a arrastrar. Este proceso puede ser muy complejo si los posibles objetos a arrastrar son muchos. La técnica común es crear un rectángulo o una región para cada objeto, y comprobar si las coordenadas del cursor están en cada uno de esos rectángulos o regiones, usando las funciones [PtInRect](#) o [PtInRegion](#).

Una vez localizado el objeto seleccionado entramos en el modo de arrastre. Esto implica algunas acciones, como capturar el ratón para la ventana actual, usando [SetCapture](#), opcionalmente, ocultar el cursor con [ShowCursor](#), ya que el objeto arrastrado puede hacer su función, y borrar el objeto de su posición inicial, ya que ahora deberá desplazarse sobre la ventana a la vez que el ratón. También necesitamos una variable estática que nos indique que estamos en una operación de arrastrar y soltar.

Finalmente tenemos que usar dos funciones específicas cuando se arrastran imágenes pertenecientes a listas de imágenes. [ImageList\\_BeginDrag](#) y [ImageList\\_DragEnter](#).

La primera función, [ImageList\\_BeginDrag](#), crea una lista de imágenes temporal que se usa para mostrar la imagen a arrastrar. Necesita cuatro parámetros: el primero es un manipulador de la lista de imágenes que contiene la imagen a arrastrar, el segundo el índice dentro de la lista que identifica la imagen. El tercero y cuarto indican las coordenadas del punto caliente. Este punto es análogo al punto activo de un cursor. De hecho, la imagen arrastrada se comporta de forma similar a cómo lo hace un cursor del ratón. Las coordenadas del punto de acceso son relativas a la esquina superior izquierda de la imagen, por lo que tendremos que hacer ciertos cálculos a partir de las coordenadas del ratón y las de la esquina superior izquierda de la imagen, concretamente, esos cálculos son una resta.

La segunda función, [ImageList\\_DragEnter](#), se encarga de bloquear las actualizaciones de la ventana especificada durante una operación de arrastre y además muestra la imagen arrastrada en la

posición especificada dentro de la ventana. Esta función requiere tres parámetros: el primero es un manipulador de la ventana en la que se va a realizar la operación de arrastre, el segundo y tercer parámetros son las coordenadas en las que se muestra la imagen, pero hay que tener cuidado, ya que esas coordenadas son relativas a la esquina superior izquierda de la ventana, y no del área de cliente. Necesitaremos, pues, el desplazamiento del área de cliente con respecto a la ventana. Podemos hacerlo mediante estas llamadas:

```
static int cxBorde;
static int cyBorde;
RECT rv;
POINT p;
...
    GetWindowRect(hwnd, &rv);
    p.x=p.y=0;
    ClientToScreen(hwnd, &p);
    cxBorde = p.x-rv.left;
    cyBorde = p.y-rv.top;
```

Primero obtenemos las coordenadas de la ventana en el rectángulo rv. El rectángulo contendrá las esquinas de la ventana en coordenadas de pantalla. Iniciamos el punto p con las coordenadas 0,0, y traducimos esas coordenadas de cliente a pantalla. El desplazamiento del área de cliente sobre la esquina superior izquierda de la ventana es la diferencia entre las coordenadas del punto y las de la esquina superior izquierda en rv.

El tratamiento del mensaje [WM\\_LBUTTONDOWN](#) queda así:

```
case WM_LBUTTONDOWN:
    ptCur.x = LOWORD(lParam);
    ptCur.y = HIWORD(lParam);
    if(!PtInRect(&re, ptCur)) return FALSE;
    // Capturar el ratón
    SetCapture(hwnd);
    endrag = TRUE;
```

```

        ShowCursor(FALSE);
        // Borrar la imagen a arrastrar:
        InvalidateRect(hwnd, &re, TRUE);
        UpdateWindow(hwnd);
        // Calcular el hotspot:
        hotspot.x = ptCur.x-re.left;
        hotspot.y = ptCur.y-re.top;
        ImageList_BeginDrag(hIml, 2, hotspot.x,
hotspot.y);
        ImageList_DragEnter(hwnd, ptCur.x + cxBorde,
ptCur.y + cyBorde);
        break;

```

## Arrastre

Durante el arrastre mantendremos pulsado el botón del ratón, y procesaremos el mensaje [WM\\_MOUSEMOVE](#). Cada vez que lo procesemos invocaremos a la función [ImageList\\_DragMove](#), al que pasaremos las coordenadas de ventana de la posición de la imagen. De nuevo, las coordenadas del ratón que obtenemos del mensaje son coordenadas de cliente, por lo que habrá que añadir el desplazamiento del área de cliente.

```

        case WM_MOUSEMOVE:
            if(!endrag) return TRUE;
            ptCur.x = LOWORD(lParam);
            ptCur.y = HIWORD(lParam);
            ImageList_DragMove(ptCur.x + cxBorde, ptCur.y +
cyBorde);
            break;

```

## Final del arrastre

Finalmente, en algún momento finalizaremos la operación de arrastrar y soltar, soltando el botón del ratón. Para ello procesaremos el mensaje [WM\\_LBUTTONDOWN](#).

Lo primero es terminar la operación de arrastre, invocando a la función [ImageList\\_EndDrag](#), que no necesita argumentos. A continuación invocaremos a la función [ImageList\\_DragLeave](#), usando como argumento un manipulador de la ventana. Esta función vuelve a permitir las actualizaciones en la ventana, borra la imagen arrastrada, y también la lista de imágenes temporal creada para la operación de arrastre.

Sólo nos queda volver a hacer visible el cursor, si lo habíamos ocultado, dibujaremos la imagen en la posición final, y liberamos el ratón.

Finalmente, actualizamos el rectángulo con la nueva posición de la imagen, de modo que podamos volver a arrastrarla.

```
case WM_LBUTTONDOWN:
    if(!endrag) return TRUE;
    ptCur.x = LOWORD(lParam);
    ptCur.y = HIWORD(lParam);
    ImageList_EndDrag();
    ImageList_DragLeave(hwnd);

    ShowCursor(TRUE);
    endrag = FALSE;

    DibujarImagen(hwnd, hIml, 2, ptCur.x - hotspot.x,
ptCur.y - hotspot.y);

    ReleaseCapture();
    re.left = ptCur.x - hotspot.x;
    re.top = ptCur.y - hotspot.y;
    re.right = re.left+32;
    re.bottom = re.top+32;
    return TRUE;
```

Hay otras dos funciones relacionadas con el arrastre de imágenes. [ImageList\\_SetDragCursorImage](#) crea una nueva imagen de arrastre mediante la combinación de la imagen especificada, generalmente la imagen de un cursor del ratón, y de la imagen actualmente arrastrada. Si se usa esta imagen generada es

imprescindible ocultar el cursor, ya que de otro modo se mostrarán dos cursores, lo que siempre es poco conveniente.

Ya hemos mencionado que [ImageList\\_BeginDrag](#) crea una lista de imágenes temporal, la función [ImageList\\_GetDragImage](#) recupera la imagen temporal, además de su posición actual y el desplazamiento del punto activo.

## Ejemplo 82

### Información de imagen

Disponemos de dos funciones para obtener información sobre listas de imágenes. La primera es [ImageList\\_GetImageInfo](#), que obtiene algunos datos sobre una imagen dentro de una lista de imágenes. Se necesitan tres parámetros, el primero es un manipulador de una lista de imágenes, el segundo el índice de la imagen sobre la que queremos información, y el tercero un puntero a una estructura [IMAGEINFO](#) en la que se almacenarán los datos sobre la imagen. Esta estructura tiene cinco campos, aunque dos de ellos no se usan:

- `hbmImage`: contiene un manipulador del mapa de bits que contiene las imágenes de la lista de imágenes.
- `hbmMask`: contiene un manipulador del mapa de bits monocromo que contiene la máscara de las imágenes de la lista de imágenes.
- `rcImage`: rectángulo que bordea la imagen indicada en el índice dentro del mapa de bits.

La otra función es [ImageList\\_GetImageCount](#) que devuelve el número de imágenes que contiene la lista de imágenes cuyo manipulador se ha pasado como parámetro.

# Capítulo 49 Ventana de estado

Las ventanas de estado son ventanas hijas que generalmente se colocan en la parte inferior de las ventanas, y que ocupan todo el ancho de la ventana padre. Se usan para mostrar mensajes de estado o de ayuda de la aplicación. Se trata de un control estático, ya que sólo se usa para mostrar información, y no para recogerla del usuario.

## Cómo crear ventanas de estado

Hay dos formas de crear ventanas de estado. La más sencilla es usar la función [CreateStatusWindow](#) a la que proporcionaremos cuatro parámetros. El primero es una combinación de estilos de ventana, entre los que debe aparecer `WS_CHILD`, y por supuesto, `WS_VISIBLE`, si queremos que la ventana se muestre. El segundo es el texto que queremos que aparezca, y que seguramente varíe a lo largo de la ejecución del programa. El tercero es un manipulador de la ventana padre y el cuarto un identificador del control, que usaremos para enviarle mensajes a la ventana de estado.

```
CreateStatusWindow(WS_CHILD|WS_VISIBLE, "Texto de prueba", hwnd, ID_STATUS);
```

La segunda forma es usar la función [CreateWindowEx](#), especificando como clase de ventana el valor `STATUSCLASSNAME`. Las coordenadas y dimensiones de la ventana serán ignoradas, de modo que pueden ser cero.

```
        CreateWindowEx(0, STATUSCLASSNAME, "Texto de
prueba", WS_CHILD|WS_VISIBLE,
                        0, 0, 0, 0, hwnd, (HMENU)ID_STATUS,
hInstance, 0);
```

Como en todos los controles comunes que estamos viendo, hay que asegurarse de que la DLL ha sido cargada invocando a la función [InitCommonControlsEx](#) indicando el valor de bandera `ICC_BAR_CLASSES` en el miembro `dw/CC` de la estructura [INITCOMMONCONTROLSEX](#) que pasaremos como parámetro.

```
INITCOMMONCONTROLSEX icce;
...
icce.dwSize = sizeof(INITCOMMONCONTROLSEX);
icce.dwICC = ICC_BAR_CLASSES;
InitCommonControlsEx(&icce);
```

## Estilos

Por defecto, las ventanas de estado se colocan en la parte inferior de la ventana padre, es decir, estas ventanas tienen activo por defecto el estilo [CCS\\_BOTTOM](#). También se puede especificar el estilo [CCS\\_TOP](#), que la coloca en la parte superior, aunque no es nada habitual.

También se puede usar el estilo `SBARS_SIZEGRIP` para incluir un mapa de bits a la derecha de la barra que se usa para redimensionar la ventana. Aunque ese estilo se puede combinar con [CCS\\_TOP](#), carece de sentido hacerlo, ya que no funcionará.

Cuando se usan barras de estado es importante que el procedimiento de ventana de la ventana padre procese el mensaje [WM\\_SIZE](#), de modo que cada vez que la ventana padre cambie de tamaño, se envíe el mensaje a la ventana de estado para que se adapte al nuevo tamaño de su ventana padre.



```
        case WM_SIZE:
            SendDlgItemMessage(hwnd, ID_STATUS, WM_SIZE, 0,
0);
            break;
```

## Ayuda para menús

Una de las aplicaciones de la ventana de estado es mostrar ayudas contextuales en función de la opción de menú actualmente seleccionada.

Para ello se usa la función [MenuHelp](#). Según la documentación del API, esta función procesa los mensajes [WM\\_MENUSELECT](#) y [WM\\_COMMAND](#).

Hay que pasar siete parámetros a la función, el primero es el mensaje, [WM\\_MENUSELECT](#) o [WM\\_COMMAND](#), seguido de los parámetros wParam y lParam del mensaje. El cuarto es un manipulador del menú de la ventana. El Quinto un manipulador de la instancia desde la que se cargarán las cadenas con los mensajes de ayuda. El sexto es un manipulador de la ventana de estado y el séptimo y último es un array de enteros con los desplazamientos de los identificadores de las cadenas con respecto a los identificadores de comandos asociados al menú.

En un menú sencillo, con un único nivel de submenús, el primer valor en el array de desplazamientos es el desplazamiento entre los identificadores de ítem de menú y sus cadenas de ayuda. Por ejemplo, si los identificadores de menú empiezan en 100, y los de las cadenas asociadas en 200, el primer valor del array será 100 (200-100). El segundo valor se usa para las cadenas de los popups, que no tienen identificador de menú. En su lugar se usa la posición, 0 para el primero, 1 para el segundo, etc. De modo que si el primer identificador de cadena para los menús popup es 800, ese será el valor del desplazamiento almacenado en el array de desplazamientos.

El array de desplazamientos contiene parejas de enteros sin signo, de modo que debe tener un número par de valores. Además, el final del array se marca con dos valores cero.

```
static UINT desplazamientos[] = {
    100, 800,
    0, 0
};
...
case WM_MENUSELECT:
    MenuHelp(msg, wParam, lParam, GetMenu(hwnd),
hInstance, GetDlgItem(hwnd, ID_STATUS), ids);
    break;
```

Si tenemos un segundo nivel de menús desplegables, tendremos que añadir una nueva pareja de valores al array de desplazamientos. El primer valor de la pareja será un desplazamiento entre identificadores de menú y de cadena y el segundo el número de orden del submenú.

Esto implica algunos problemas si, por ejemplo, nuestro menú tiene dos opciones de menú horizontal, y en la primera posición del segundo hay un submenú popup. El desplazamiento de ese segundo submenú es 0:

```
LANGUAGE LANG_NEUTRAL, SUBLANG_NEUTRAL
menu MENU
{
    POPUP "Principal"
    {
        MENUITEM "Nuevo", IDM_NUEVO
        MENUITEM "Abrir...", IDM_ABRIR
        MENUITEM "Guardar", IDM_GUARDAR
        MENUITEM "Guardar como...", IDM_GUARDAR_COMO
        MENUITEM SEPARATOR
        MENUITEM "Salir", IDM_SALIR
    }
    POPUP "Ver"
    {
        POPUP "Tamaño de letra"
```

```

        {
            MENUITEM "Pequeña", IDM_PEQUE
            MENUITEM "Mediana", IDM_MEDIANA
            MENUITEM "Grande", IDM_GRANDE
        }
        MENUITEM "Barra", IDM_BARRA
    }
}

LANGUAGE LANG_SPANISH, SUBLANG_SPANISH_MODERN
STRINGTABLE
{
    IDS_NUEVO                "Nuevo fichero"
    IDS_ABRIR                "Abrir un fichero existente"
    IDS_GUARDAR              "Guardar fichero actual"
    IDS_GUARDAR_COMO         "Guardar una copia del
fichero actual con otro nombre"
    IDS_SALIR                "Salir de la aplicación"
    IDS_BARRA                "Mostrar u ocultar barra de
estado"
    IDS_PEQUE                "Establece un tamaño de
letra pequeño"
    IDS_MEDIANA              "Establece un tamaño de
letra normal"
    IDS_GRANDE              "Establece un tamaño de
letra grande"
    IDS_PRINCIPAL            "Menú principal"
    IDS_VER                  "Opciones de visualización"
    IDS_TAMANO               "Opciones para el tamaño de
letra"
}

```

Para esta estructura de menú, los valores del array de desplazamientos son:

```

static UINT desplazamientos[] = {
    100, 800,
    802, 0,
    0, 0
};

```

Pero, tal como está diseñada la función, se cargará la misma cadena para todos los menús desplegables con desplazamiento

cero, es decir, para el menú principal y para el de tamaño de letra.

Aunque hay algunas cosas que podemos hacer para que funcione correctamente con nuestros menús, las soluciones serán engorrosas y posiblemente no funcionen en futuras versiones del API. Lo más habitual es no mostrar textos de ayuda para los menús popup, sino sólo para las opciones de ítems de menú.

**Nota:**

Además, esta función tiene otras limitaciones que desaconsejan o limitan su uso en nuestros programas, por ejemplo, no es fácil que funcione correctamente con menús creados en la ejecución, ya que las cadenas se cargan desde recursos de cadena. Todas las cadenas se cargan desde la misma instancia, de modo que para los menús creados desde varias dll no funcionará bien.

## Ejemplo 83

### Tamaño y altura

Generalmente, las dimensiones de las barras de estado se ajustan automáticamente por su procedimiento de ventana, ignorando cualquier tamaño indicado por la aplicación. La anchura es la misma que la del área de cliente de la ventana padre, y la altura se ajusta en función del tamaño de fuente seleccionada por el contexto de dispositivo de la ventana de estado.

Por eso es importante, como ya comentamos antes, que la ventana padre procese el mensaje [WM\\_SIZE](#), y lo reenvíe a la ventana de estado.

Sin embargo, la aplicación puede asignar la altura mínima de la ventana de estado, o más concretamente, del área útil de la ventana

de estado, descontando los bordes. Esto se hace con el mensaje `SB_SETMINHEIGHT`, indicando en `wParam` la altura mínima, en pixels. Generalmente esto sólo será necesario en barras de estado owner-draw.

También podemos recuperar la dimensiones de los bordes de una barra de estado mediante el mensaje `SB_GETBORDERS`. En el parámetro `lParam` tendremos que pasar la dirección de un array de tres enteros en los que recibiremos las anchuras del borde horizontal, vertical y el la distancia entre los rectángulos interiores que contienen el texto, respectivamente.

## Ventanas de estado con varias partes

Podemos dividir la barra de estado en partes de diferente anchura, cada una de las cuales puede mostrar un texto diferente, que usaremos para informar al usuario sobre el estado de ciertos valores de la aplicación.

Para dividir la barra de estado en partes se usa un mensaje `SB_SETPARTS`, indicando en el parámetro `wParam` el número de partes y el `lParam` la dirección de un array con tantos valores de enteros como partes, cada uno de los cuales es la distancia, desde el borde izquierdo de la ventana al borde derecho de cada parte, en pixels y en coordenadas de cliente.

Para que la barra de estado ocupe todo el ancho de la ventana padre es necesario que el valor de anchura para la última parte coincida con la anchura del área de cliente. También se puede usar el valor -1 para esa última anchura, de modo que la última parte se ensanchará automáticamente hasta ocupar toda la anchura disponible.

```
int anchura[] = {54, 74, 94, -1};  
...  
CreateStatusWindow(WC_CHILD|WS_VISIBLE,  
"Texto de prueba", hwnd, ID_STATUS);
```

```
SendDlgItemMessage(hwnd, ID_STATUS,  
SB_SETPARTS, 4, (LPARAM)anchura);
```

Pero probablemente estemos acostumbrados a que la barra de estado ocupe todo el ancho de la ventana, y que la parte que ajusta la anchura sea la primera, y no la última, siendo el resto de las partes siempre igual de anchas.

Para conseguir ese efecto aprovecharemos el procesamiento del mensaje `WM_SIZE`, obtendremos la anchura del área de cliente, y ajustaremos las anchuras de cada parte en función de esa anchura.

```
RECT re;  
int anchura[4];  
...  
case WM_SIZE:  
    GetClientRect(hwnd, &re);  
    anchura[3] = re.right-20;  
    anchura[2] = anchura[3]-60;  
    anchura[1] = anchura[2]-60;  
    anchura[0] = anchura[1]-60;  
    SendDlgItemMessage(hwnd, ID_STATUS, SB_SETPARTS,  
4, (LPARAM)anchura);  
    SendDlgItemMessage(hwnd, ID_STATUS, msg, wParam,  
lParam);
```

El número de partes en las que se puede dividir una ventana de estados está limitado a 255. Se trata de un límite poco importante, ya que generalmente usaremos muchas menos partes.

También es posible obtener las dimensiones actuales de las partes de una ventana de estado, usando un mensaje `SB_GETPARTS`. En `wParam` indicaremos el número de partes a recuperar, si ese número es mayor que el número de partes existentes, sólo se recuperarán las que existan. En `lParam` pasaremos un puntero a un array de enteros en los que recibiremos las distancias de los bordes derechos de cada parte. El valor de retorno es el número de partes cuyas anchuras son recuperadas. Si

se usa el valor cero para lParam, recuperaremos sólo el número de partes.

```
int nPartes;  
int *anchura;  
anchura = (int*)malloc(sizeof(int)*10);  
nPartes = SendDlgItemMessage(hwnd,  
ID_STATUS, SB_GETPARTS, 10, anchura));  
free(anchura);
```

## Manejar texto

En cada parte de una ventana de estado se puede modificar el texto o recuperarlo.

Para modificarlo se usa el mensaje [SB\\_SETTEXT](#). En el parámetro wParam se indica el índice, empezando en cero, de la parte en que se quiere modificar el texto, combinado con el tipo de texto a mostrar. Ese tipo puede ser 0, que indica que se trace un borde hundido, SBT\_POPOUT que indica un borde sobresaliente, SBT\_NOBORDERS que indica que no se tracen bordes, SBT\_OWNERDRAW para ventanas de estado owner-draw o SBT\_RTLREADING para lenguajes que se escriben de derecha a izquierda.

En lParam se pasa un puntero a la cadena terminada con cero que se debe mostrar, o un valor entero de 32 bits, en caso de ventanas de estado owner-draw.

```
SendDlgItemMessage(hwnd, ID_STATUS, SB_SETTEXT,  
3|SBT_POPOUT, (LPARAM) "\tbloq num");
```

Por otra parte, podemos usar caracteres tabuladores ('\t') para modificar la posición del texto. Si no se usa ninguno, la cadena se alinea a la izquierda de la parte indicada. El texto a la derecha de un

tabulador se muestra centrado en la parte indicada, y el texto después del segundo tabulador se muestra alineado a la derecha.

Para recuperar texto desde una parte de una ventana de estado usaremos el mensaje [SB\\_GETTEXT](#), y para calcular la longitud del texto en una parte, [SB\\_GETTEXTLENGTH](#).

En el mensaje [SB\\_GETTEXT](#) indicaremos en el parámetro `wParam` el índice, basado en cero, de la parte cuyo texto queremos recuperar y en `lParam` un puntero a un buffer que recibirá la cadena. El valor de retorno indicará el tipo de borde usado para trazar el texto en la palabra de mayor peso, y la longitud de la cadena recuperada en la de menor peso.

En el mensaje [SB\\_GETTEXTLENGTH](#) sólo usaremos el parámetro `wParam` para indicar el índice de la parte. El valor de retorno será similar al del mensaje [SB\\_GETTEXT](#).

```
char *txt;
int len;
...
    len = LOWORD(SendDlgItemMessage(hwnd, ID_STATUS,
SB_GETTEXTLENGTH, 3, 0));
    txt = (char*)malloc(len+1);
    SendDlgItemMessage(hwnd, ID_STATUS, SB_GETTEXT,
3, (LPARAM)txt);
...
    free(txt);
```

Si se trata de una ventana de estado con un única parte, se pueden usar los mensajes [WM\\_SETTEXT](#), [WM\\_GETTEXT](#) y [WM\\_GETTEXTLENGTH](#), tratando la ventana de estado como un simple control de texto estático.

Finalmente, hay otra posibilidad de mostrar texto en una ventana de estado sin necesidad de crearla, usando la función [DrawStatusText](#). Aunque en realidad lo que muestra se parece más a un control de texto estático.

El primer parámetro es un manipulador del DC de la ventana, el segundo un puntero a una estructura [RECT](#) con las coordenadas de



la zona ocupada por el texto. Este rectángulo se usa para mostrar los bordes y la posición del texto. El tercer parámetro es el texto a mostrar, los tabuladores funcionan de forma similar a como lo hacen en el mensaje [SB\\_SETTEXT](#). El cuarto parámetro indica el tipo de bordes a trazar.

```
HDC hdc;  
RECT re;  
  
hdc = GetDC(hwnd);  
re.left = 20; re.top = 10;  
re.right = 150; re.bottom = 30;  
DrawStatusText(hdc, &re, "\tTexto estático",  
SBT_POPOUT);  
ReleaseDC(hwnd, hdc);
```

## Ejemplo 84

### Ventanas de estado owner-draw

Cada una de las partes de una ventana de estado se puede definir como owner-draw. Esto proporciona un mayor control sobre los contenidos de esas partes, ya que nos permite incluir mapas de bits, o en general, cualquier gráfico GDI que queramos, en lugar de sólo texto.

Para definir una parte como owner-draw basta con enviar un mensaje [SB\\_SETTEXT](#), añadiendo el tipo SBT\_OWNERDRAW al identificador de la parte, en el parámetro wParam, y usando el parámetro lParam para un valor de 32 bits que posteriormente se usará para dibujar el contenido de la parte. Ese parámetro puede ser un puntero a una estructura, un mapa de bits, un entero, etc. El significado del valor queda definido por la aplicación, es decir, por nosotros.

```

        SendDlgItemMessage(hwnd, ID_STATUS, SB_SETTEXT,
2|SBT_OWNERDRAW, lParam);
...
        SendDlgItemMessage(hwnd, ID_STATUS, SB_SETTEXT,
3|SBT_OWNERDRAW, (LPARAM)hBitmapSi);

```

Cada vez que la aplicación necesita actualizar el contenido de una parte owner-draw de una ventana de estado, se envía un mensaje [WM\\_DRAWITEM](#) a la ventana padre. El parámetro wParam del mensaje contiene el identificador de ventana de la ventana de estado, y el parámetro lParam es un puntero a una estructura [DRAWITEMSTRUCT](#). Toda la información necesaria para dibujar el contenido de la parte está incluida en esta estructura.

Cuando el mensaje [WM\\_DRAWITEM](#) es recibido para mostrar una parte de una barra de estado owner-draw el significado de algunos campos de la estructura [DRAWITEMSTRUCT](#) es algo diferente de la que se explica en la documentación:

Miembro	Descripción
CtlType	No definido, no se usa.
CtlID	Identificador de la barra de estado.
itemID	Índice de la parte a dibujar.
itemAction	No definido, no se usa.
itemState	No definido, no se usa.
hwndItem	Manipulador de la ventana de estado.
hDC	Manipulador del contexto de dispositivo de la ventana de estado.
rcItem	Coordenadas de la parte de la ventana a dibujar. Estas coordenadas son relativas a la esquina superior izquierda de la ventana de estado.
itemData	Valor de 32 bits definido por la aplicación especificado mediante el parámetro lParam del mensaje <a href="#">SB_SETTEXT</a> .

```

DRAWITEMSTRUCT *dis;
HDC memDC;
POINT ptCur;
char cad[40];
...
case WM_DRAWITEM:
    dis = (DRAWITEMSTRUCT*)lParam;
    switch(dis->itemID) {
        case 2:
            memDC = CreateCompatibleDC(dis->hDC);
            SelectObject(memDC, hBitmapCoor);
            ptCur.x = LOWORD(dis->itemData);
            ptCur.y = HIWORD(dis->itemData);
            sprintf(cad, "%d,%d", ptCur.x, ptCur.y);
            SetBkMode(dis->hDC, TRANSPARENT);
            TextOut(dis->hDC, dis->rcItem.left+20, dis->rcItem.top, cad, strlen(cad));
            BitBlt(dis->hDC, dis->rcItem.left, dis->rcItem.top, 16, 16, memDC, 0, 0, SRCCOPY);
            DeleteDC(memDC);
            break;
        case 3:
            memDC = CreateCompatibleDC(dis->hDC);
            SelectObject(memDC, (HBITMAP)dis->itemData);
            BitBlt(dis->hDC, dis->rcItem.left, dis->rcItem.top, 16, 16, memDC, 0, 0, SRCCOPY);
            DeleteDC(memDC);
            break;
        default:
            break;
    }
    break;

```

En este ejemplo vemos cómo utilizamos los valores de la estructura **DRAWITEMSTRUCT** para discriminar la parte a dibujar, seleccionar el DC de la ventana de salida, obtener las coordenadas y el valor del parámetro. En el caso de la parte 2, el parámetro son las coordenadas del ratón, en la palabra de menor peso la coordenada x y en la de mayor peso la coordenada y. En el caso de la parte 3, lParam contiene un manipulador de mapa de bits.

## Ejemplo 85

### Ventanas de estado simples

Siempre se puede cambiar el tipo de ventana de estado al modo simple (con una única parte), mediante un mensaje [SB\\_SIMPLE](#). El texto asignado a la ventana de estado simple se mantiene almacenado de forma independiente del de las partes cuando no está en modo simple, de modo que se puede cambiar de un modo a otro sin que se pierdan los contenidos de ninguna parte. Es habitual que los textos de ayuda de los menús se muestren en modo simple, y una vez el menú ha perdido el foco, se vuelva al modo no simple, sin necesidad de actualizar cada parte.

La única limitación a tener en cuenta en ventanas de estado simples es que no adminten el modo owner-draw.

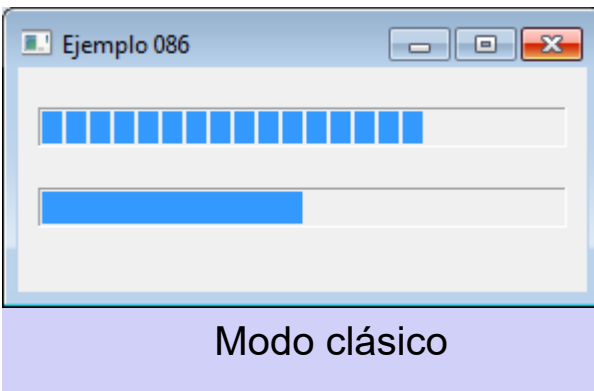
# Capítulo 50 Barra de progreso

Podemos considerar las barras de progreso como un control estático, dado que no sirven para obtener datos desde el usuario, sino sólo para informarle sobre el estado del progreso de una tarea, generalmente de una duración considerable.

De nuevo, hay que asegurarse de que la DLL ha sido cargada invocando a la función [InitCommonControlsEx](#) indicando el valor de bandera `CC_PROGRESS_CLASS` en el miembro `dwICC` de la estructura [INITCOMMONCONTROLSEX](#) que pasaremos como parámetro.

```
INITCOMMONCONTROLSEX icCE;  
...  
icCE.dwSize = sizeof(INITCOMMONCONTROLSEX);  
icCE.dwICC = CC_PROGRESS_CLASS;  
InitCommonControlsEx(&icCE);
```

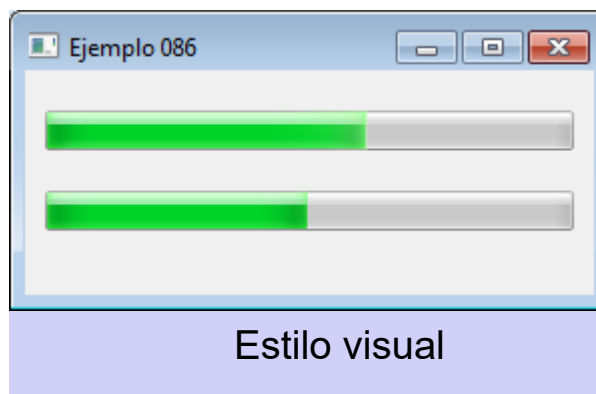
## Estilos visuales



A partir de la versión XP de Windows es posible seleccionar diferentes temas que definen muchos aspectos visuales del interfaz de Windows. Entre ellos, algunos afectan a la apariencia de las barras de progreso.

En nuestros programas podemos optar por una apariencia clásica (modo clásico) o por una apariencia más actual (estilos visuales).

Por defecto se usará el modo clásico, en el que las barras se suelen mostrar en color azul sobre fondo blanco, si efectos 3D. En modo clásico los mensajes para cambiar los colores de fondo y de la barra funcionan como se espera, de modo que aunque visualmente pueden ser menos atractivos, nos proporcionan mayor control sobre la apariencia de estos controles.



Si se activan los estilos visuales, la apariencia de los controles es más atractiva: se añaden efectos 3D, que aparentan relieve, y animaciones de color. Pero perdemos la posibilidad de modificar los colores desde la aplicación, ya que esos parámetros dependen sólo del estilo activo. Tampoco podemos diferenciar entre las barras de progreso de bloques o suaves, como en el modo clásico.

Para activar los estilos visuales tenemos dos alternativas, y algunas condiciones previas.

Las condiciones están relacionadas con la versión de sistema operativo y de Internet Explorer instaladas, o más concretamente, con la versión de la DLL de los controles comunes, que generalmente se asocia con una versión de Internet Explorer.

El sistema operativo debe ser Windows XP o posterior, y la de la DLL la 6.0 o posterior. Para que el compilador tenga esto en cuenta hay que definir estas macros:

```
#define _WIN32_WINNT 0x0501
#define WINVER 0x0501
#define _WIN32_IE 0x0600
```

Aunque lo normal es que estén definidas si tu compilador fue instalado en Windows XP o un sistema operativo posterior, y no necesitamos declararlas.

La otra condición es agregar al programa un fichero de *manifiesto*, en el que se indican las versiones y dependencias de DLLs, entre otras cosas.

Aquí es donde aparecen dos opciones: incluir el fichero de manifiesto en un fichero independiente, cuyo nombre será <nombre\_aplicación>.exe.manifest, donde <nombre\_aplicación> debe ser el nombre del fichero ejecutable de la aplicación. Ese fichero debe estar en la misma carpeta que el programa ejecutable.

La segunda opción es incluir el manifiesto en el fichero de recursos, de modo que se enlace con el programa ejecutable, y formen un único fichero.

Cada opción tiene sus ventajas e inconvenientes. Si se usa un fichero independiente siempre tendremos la opción de borrarlo o renombrarlo, de modo que se active el modo clásico automáticamente. Sin embargo, si queremos mantener el control sobre el aspecto de la aplicación, esta opción tiene el inconveniente de que el usuario puede borrar el fichero o que puede resultar corrupto.

En cuanto a incluirlo en el fichero de recursos, la limitación es que el modo clásico deja de ser accesible, a no ser que se recompile la aplicación.

## Fichero de manifiesto

Los ficheros de manifiesto son ficheros de texto en formato XML, con el siguiente contenido:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0">
  <assemblyIdentity
    version="1.0.0.0"
    processorArchitecture="*"
    name="CompanyName.ProductName.YourApplication"
    type="win32"
  />
```

```
<description>Your application description here.
</description>
<dependency>
  <dependentAssembly>
    <assemblyIdentity
      type="win32"
      name="Microsoft.Windows.Common-Controls"
      version="6.0.0.0"
      processorArchitecture="*"
      publicKeyToken="6595b64144ccf1df"
      language="*"
    />
  </dependentAssembly>
</dependency>
</assembly>
```

La parte que nos interesa, para activar los estilos visuales es la de `<dependency>`, que indica que se use la versión 6.0 de la DLL de controles comunes. Las claves de `<assemblyIdentity>` y `<description>` podemos ajustarlas dependiendo de nuestra aplicación, indicando la versión, los datos del programador o la empresa que realiza el programa, y la descripción de la aplicación.

Si optamos por la primera opción, bastará salvar un fichero con este contenido en la misma carpeta que el ejecutable, y con el nombre indicado.

## Manifiesto en fichero de recursos

En el caso de incluir en manifiesto en el fichero de recursos, bastará con añadir estas líneas:

```
LANGUAGE LANG_NEUTRAL, SUBLANG_NEUTRAL
1          RT_MANIFEST      "..\\manifest.xml"
```

Por supuesto, necesitaremos un fichero externo, en este caso con el nombre "manifest.xml", y con un contenido análogo al anterior, aunque algo diferente:



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0">
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
        type="win32"
        name="Microsoft.Windows.Common-Controls"
        version="6.0.0.0"
        processorArchitecture="*"
        publicKeyToken="6595b64144ccf1df"
        language="*"
      />
    </dependentAssembly>
  </dependency>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel
          level="asInvoker"
          uiAccess="false"/>
        </requestedPrivileges>
      </security>
    </trustInfo>
  </assembly>
```

## Estilos

Hay dos estilos disponibles para las barras de progreso.

El primero, `PBS_VERTICAL`, afecta a la orientación. Si se especifica, la barra se rellenará de abajo a arriba, si no se especifica, se rellenará de izquierda a derecha.

El segundo, `PBS_SMOOTH`, afecta a la apariencia. Si se especifica el relleno de la barra será continuo y no habrá cortes ni saltos. Si no se especifica, la barra se divide en rectángulos. Este estilo no tiene efecto cuando usamos estilos visuales.

## Cómo crear barras de progreso

Como siempre, podemos insertar barras de progreso en nuestras ventanas mediante la función [CreateWindowEx](#), especificando como clase de ventana el valor `PROGRESS_CLASS`:

```
CreateWindowEx(0, PROGRESS_CLASS, (LPSTR)NULL,  
WS_CHILD | WS_VISIBLE, 10, 20, 200, 20, hwnd,  
            (HMENU)ID_PROGRESSBAR, hInstance, NULL);
```

También podemos incluirlas en nuestros cuadros de diálogo, mediante ficheros de recursos, como un control de la clase `PROGRESS_CLASS`:

```
LANGUAGE LANG_NEUTRAL, SUBLANG_NEUTRAL  
IDD_DIALOG1 DIALOG 0, 0, 186, 47  
STYLE DS_3DLOOK | DS_CENTER | DS_MODALFRAME | DS_SHELLFONT |  
WS_CAPTION | WS_VISIBLE | WS_POPUP | WS_SYSMENU  
CAPTION "Dialog"  
FONT 8, "Ms Shell Dlg"  
{  
    CONTROL        "", IDC_PROGRESO, PROGRESS_CLASS,  
PBS_SMOOTH, 8, 8, 169, 11  
    PUSHBUTTON     "+", IDC_MAS, 7, 24, 14, 14  
    PUSHBUTTON     "-", IDC_MENOS, 29, 24, 14, 14  
    DEFPUSHBUTTON  "OK", IDOK, 127, 25, 50, 14  
}
```

Entre los estilos, sólo hay dos disponibles: `PBS_SMOOTH` y `PBS_VERTICAL`. Se puede especificar uno de ellos, los dos o ninguno.

## Rangos

Cada barra de progreso tiene ciertos parámetros que podemos modificar a nuestra conveniencia. Los principales son los que definen el rango: valor mínimo y máximo, y el valor actual.

Cuando el valor actual sea igual al valor mínimo del rango, la barra se mostrará vacía. Cuando el valor actual sea igual al valor máximo del rango, la barra se mostrará llena.

En nuestros programas haremos variar el valor actual desde el mínimo al máximo para reflejar el progreso de alguna tarea cuando nos interese informar al usuario a medida que esa tarea se va completando.

Por defecto, el rango de las barras de progreso es (0, 100), pero podemos variar ese rango, siempre que tengamos en cuenta dos reglas sencillas y lógicas:

1. Que ambos valores deben ser positivos
2. Que el valor máximo debe ser mayor que el mínimo

Para establecer un nuevo rango disponemos de dos mensajes: [PBM\\_SETRANGE](#) y [PBM\\_SETRANGE32](#), dependiendo de si los valores máximo y mínimo son enteros de 16 ó 32 bits, respectivamente.

Generalmente, con la versión de 16 bits será suficiente, pero es bueno recordar que, en caso necesario, podemos usar valores de 32 bits para establecer los rangos.

Si optamos por la versión de 16 bits, usaremos el parámetro `lParam` para empaquetar los dos rangos, en la palabra de menor peso el rango mínimo, y en la de mayor peso, el máximo. Podemos usar la macro [MAKELPARAM](#) para empaquetar los dos valores:

```
SendDlgItemMessage(hwnd, ID_PROGRESSBAR,  
PBM_SETRANGE, 0, MAKELPARAM(0, 200));
```

Si preferimos usar la versión de 32 bits, indicaremos el rango mínimo en `wParam` y el máximo en `lParam`.

```
SendDlgItemMessage(hwnd, ID_PROGRESSBAR,
```

```
PBM_SETRANGE32, 0, 200);
```

En los dos casos, el valor de retorno es un DWORD, cuya palabra de menor peso contiene el rango mínimo previo, y la de mayor peso el rango máximo previo. Si los valores de rango previos eran de 32 bits, sólo obtendremos las palabras de menor peso de cada rango.

También podemos obtener los valores actuales de los rangos mediante un mensaje [PBM\\_GETRANGE](#). En wParam usaremos el valor TRUE para que el mensaje retorne el valor del rango mínimo, y FALSE para que retorne el máximo. En lParam podemos pasar un puntero a una estructura [PBRANGE](#), que recibirá los dos valores del rango, o NULL si no queremos usar esa estructura.

```
PBRANGE rango;

min = SendDlgItemMessage(hwnd, ID_PROGRESSBAR,
PBM_GETRANGE, TRUE, 0);
max = SendDlgItemMessage(hwnd, ID_PROGRESSBAR,
PBM_GETRANGE, FALSE, (LPARAM)&rango);
```

## Posicion

Hay tres modos de modificar la posición actual de un control de barra de progreso, podremos elegir la que más se ajuste a nuestro problema particular.

La más sencilla es asignar la posición directamente al valor que queramos, para ello podemos usar el mensaje [PBM\\_SETPOS](#), indicando en el parámetro wParam la nueva posición deseada.

```
SendDlgItemMessage(hwnd, ID_PROGRESSBAR, PBM_SETPOS,
(WPARAM)val, 0);
```

Otro modo es incrementar la posición actual en una cantidad. Esa cantidad puede ser positiva o negativa. Para hacer esto usaremos el mensaje [PBM\\_DELTAPOS](#), indicando en wParam la cantidad a incrementar.

```
SendDlgItemMessage(hwnd, ID_PROGRESSBAR, PBM_DELTAPOS,  
(WPARAM) 1, 0);
```

La última forma es establecer un paso de incremento, y hacer incrementos en la posición usando ese valor. Las barras de progreso integran un mecanismo para hacer esto. Podemos establecer el paso mediante un mensaje [PBM\\_SETSTEP](#), indicando en wParam el valor del paso, y posteriormente hacer incrementos de ese paso, enviando mensaje [PBM\\_STEPIT](#).

```
SendDlgItemMessage(hwnd, ID_PROGRESSBAR, PBM_SETSTEP,  
(WPARAM) 5, 0);  
...  
SendDlgItemMessage(hwnd, ID_PROGRESSBAR, PBM_STEPIT, 0,  
0);
```

Si necesitamos obtener el valor actual de la posición de una barra de progreso, podemos hacerlo usando un mensaje [PBM\\_GETPOS](#), sin parámetros.

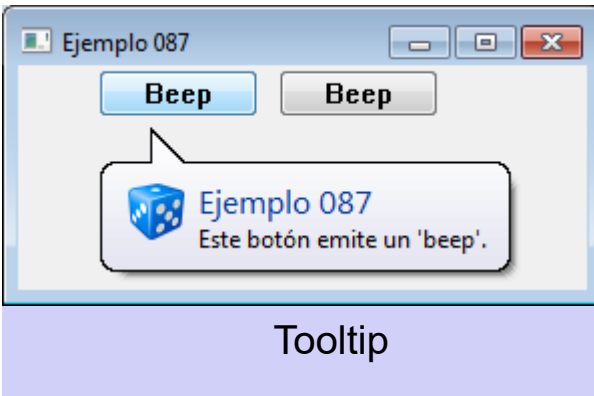
## Colores

Si usamos barras de progreso en modo clásico, podemos modificar los colores para la barra y el fondo, usando los mensajes [PBM\\_SETBARCOLOR](#) y [PBM\\_SETBKCOLOR](#), respectivamente. En controles con estilos visuales, tal como comentamos antes, estos mensajes no tienen efecto.

```
    SendDlgItemMessage(hwnd, ID_PROGRESSBAR,  
PBM_SETBARCOLOR, 0, (LPARAM)RGB(255,120,90));  
    SendDlgItemMessage(hwnd, ID_PROGRESSBAR, PBM_SETBKCOLOR,  
0, (LPARAM)RGB(0,0,0));
```

## Ejemplo 86

# Capítulo 51 Control Tooltip



Los controles Tooltip son pequeñas ventanas que aparecen sobre otros controles, generalmente llamados herramientas, y que desaparecen al cabo de un tiempo corto. Normalmente se usan para mostrar pistas o

ayudas al usuario sobre la tarea asignada al control, o para mostrar el texto completo de un ítem de una lista o árbol cuando parte de ese texto queda oculto tras el borde del control.

Cuando veamos las barras de herramientas veremos que se trata de conjuntos de botones, cada uno de los cuales indica la acción mediante un icono. Esto hace que a menudo no sea evidente qué hace cada herramienta. En estos casos, los tooltips serán especialmente útiles.

De nuevo estamos ante controles estáticos, en el sentido de que no sirven para obtener datos por parte del usuario, sino para mostrarle información.

De forma similar a lo que pasa con las listas de imágenes, veremos que estos controles se usan conjuntamente con otros que veremos en próximos capítulos, como list-views, tree-views, barras de herramientas, etc.

## Creación de tooltip

Los controles tooltip se crean usando la función [CreateWindowEx](#), indicando como tipo de ventana la constante `TOOLTIPS_CLASS`, el estilo extendido `WS_EX_TOOLWINDOW`, al

menos el estilo `WS_POPUP`, más los estilos específicos de controles tooltip que queramos.

En realidad, los controles tooltip siempre tienen los estilos `WS_EX_TOOLWINDOW` y `WS_POPUP`, aunque no se especifiquen de forma explícita.

Para las coordenadas y dimensiones usaremos la constante `CW_USEDEFAULT`, puesto que la posición dependerá de la posición del ratón y del control al que pertenezca el tooltip, y su tamaño del contenido que le asignemos.

```
        hwndTip = CreateWindowEx(WS_EX_TOOLWINDOW,  
    TOOLTIPS_CLASS, NULL,  
                                WS_POPUP | TTS_ALWAYSTIP |  
    TTS_BALLOON,  
                                CW_USEDEFAULT, CW_USEDEFAULT,  
                                CW_USEDEFAULT, CW_USEDEFAULT,  
    hwnd, NULL,  
    hInstance, NULL);
```

Inicialmente, el control tooltip estará oculto, por eso no especificamos el estilo `WS_VISIBLE`. De todos modos, aunque lo indiquemos, no se activará.

También es conveniente hacer que la ventana tooltip sea la "topmost", de modo que nunca sea ocultada por otras ventanas o controles:

```
        SetWindowPos(hwndTip, HWND_TOPMOST, 0, 0, 0, 0,  
    SWP_NOMOVE | SWP_NOSIZE | SWP_NOACTIVATE);
```

Es conveniente conservar el manipulador de ventana del tooltip en una variable estática, sobre todo si necesitamos enviarle mensajes después de crear la ventana.

Esto es necesario porque al tratarse de una ventana con el estilo `WS_POPUP`, el décimo parámetro se trata como un manipulador de



menú, y no como un identificador de control, por lo que debe ser cero. Esto hace que no podamos enviarle mensajes usando [SendDlgItemMessage](#), y necesitaremos el manipulador de ventana del tooltip.

Como con el resto de los controles comunes, es necesario asegurarse de que la DLL ha sido cargada mediante una llamada a la función [InitCommonControlsEx](#) indicando el valor de bandera `ICC_TAB_CLASSES` en el miembro `dw/CC` de la estructura [INITCOMMONCONTROLSEX](#) que pasaremos como parámetro.

```
INITCOMMONCONTROLSEX icCE;  
...  
icCE.dwSize = sizeof(INITCOMMONCONTROLSEX);  
icCE.dwICC = ICC_TAB_CLASSES;  
InitCommonControlsEx(&icCE);
```

## Estilos

Los controles tooltip tienen varios [estilos](#) propios que controlan varios aspectos gráficos:

- `TTS_ALWAYSSTIP`: el control tooltip se activa aunque la ventana padre no tenga el foco.
- `TTS_BALLOON`: tooltip en forma de globo, con las esquinas redondeadas y un saliente que apunta al control al que pertenece.
- `TTS_CLOSE`: añade un botón de cerrar, es necesario que también esté activo el estilo `TTS_BALLOON` y que se haya asignado un título, además deben estar activos los estilos visuales.
- `TTS_NOANIMATE`: desactiva la animación inicial del control, que hace que parezca que la ventana se desenrolla.
- `TTS_NOFADE`: desactiva la animación final, que hace que la ventana se desvanezca.

- TTS\_NOPREFIX: evita que se eliminen los caracteres '&' y que la cadena se de por terminada cuando se encuentre el primer carácter tabulador.
- TTS\_USEVISUALSTYLE: está relacionado con hiperenlaces, y no lo veremos en este capítulo.

## Activar y desactivar tooltips

Por defecto, una vez creado, el tooltip quedará activado, pero podemos desactivarlo y volverlo a activar mediante el mensaje [TTM\\_ACTIVATE](#), indicando en wParam un valor TRUE para activarlo o FALSE para desactivarlo.

```
SendMessage(hwndTip, TTM_ACTIVATE, (WPARAM)TRUE, 0);
```

## Cambios de color

Es posible modificar los colores del fondo y de los caracteres de un control tooltip, usando los mensajes [TTM\\_SETTIPBKCOLOR](#) y [TTM\\_SETTIPTEXTCOLOR](#), respectivamente. En ambos casos, el color deseado se especifica en el parámetro wParam.

```
SendMessage(hwndTip, TTM_SETTIPTEXTCOLOR,  
(WPARAM)RGB(255,0,0), 0);  
SendMessage(hwndTip, TTM_SETTIPBKCOLOR,  
(WPARAM)RGB(240,255,255), 0);
```

Hay que tener en cuenta que, como pasaba con las barras de progreso, si se activan los controles visuales, estos mensajes no tendrán efecto, y se usarán siempre los colores del tema.

## Asignar título e icono

A cada ventana tooltip se le puede asociar un icono y un título. Hay que asignar ambos o ninguno, los dos van unidos, aunque el icono que indiquemos puede ser nulo.

Para hacerlo disponemos del mensaje `TTM_SETTITLE`, en el que usaremos el parámetro `wParam` para indicar el icono, y el parámetro `lParam` para indicar el texto del título:

```
HICON hIcon = LoadIcon(hInstance, "Icono");  
SendMessage(hwndTip, TTM_SETTITLE, (WPARAM)hIcon,  
(LPARAM)"Ejemplo 087");  
DestroyIcon(hIcon);
```

Como icono podemos usar uno de los iconos predefinidos, para lo que disponemos de varias constantes:

- `TTI_NONE`: Sin icono.
- `TTI_INFO`: Icono de información.
- `TTI_WARNING`: Icono de aviso.
- `TTI_ERROR`: Icono de error.
- `TTI_INFO_LARGE`: Icono de información grande.
- `TTI_WARNING_LARGE`: Icono de aviso grande.
- `TTI_ERROR_LARGE`: Icono de error grande.

Los iconos grandes sólo están disponibles si se activan los estilos visuales, con el estilo clásico sólo se pueden especificar los iconos pequeños, o `TTI_NONE`.

Si se activan los estilos visuales, en `wParam` podemos especificar cualquier icono, mediante su manipulador.

**Nota:**

la versión actual de "commctrl.h" cuando escribo este capítulo tiene algunos errores con respecto a este tema concreto. No están definidas las constantes "TTI\_", ni el mensaje "TTM\_SETTITLE". Para poder utilizar títulos de

tooltips en nuestros programas deberemos añadir las siguientes definiciones:

```
#ifndef TTM_SETTITLE
#ifdef UNICODE
#define TTM_SETTITLE TTM_SETTITLEW
#else
#define TTM_SETTITLE TTM_SETTITLEA
#endif
#endif

#ifndef TTI_NONE
// ToolTip Icons (Set with
TTM_SETTITLE)
#define TTI_NONE 0
#define TTI_INFO 1
#define TTI_WARNING 2
#define TTI_ERROR 3
#if (_WIN32_WINNT >= 0x0600)
#define TTI_INFO_LARGE 4
#define TTI_WARNING_LARGE 5
#define TTI_ERROR_LARGE 6
#endif // (_WIN32_WINNT >= 0x0600)
#endif
```

## Limitar anchura

Podemos limitar la anchura máxima del control tooltip, de modo que el texto que incluye se fragmente en distintas líneas si supera la anchura establecida. Para limitar la anchura máxima se usa el mensaje [TTM\\_SETMAXTIPWIDTH](#), indicando el lParam la anchura máxima en pixels, o -1 para permitir cualquier anchura.

```
SendMessage(hwndTip, TTM_SETMAXTIPWIDTH, 0, 150);
```

---

## Asignar a herramienta

Cada control tooltip puede ser asignado a ninguna, a una o a varias herramientas, o controles. Además, una vez asignada una herramienta, también puede ser eliminada.

Para añadir herramientas se usa el mensaje [TTM\\_ADDTOOL](#), indicando en *lParam* un puntero a una estructura [TOOLINFO](#), que contiene los datos necesarios para añadir un control o un rectángulo a un tooltip.

Algunos de los campos de la estructura [TOOLINFO](#) es obligatorio especificarlos. Principalmente, el primero de ellos, *cbSize* que contiene el tamaño de la estructura. A ese campo le asignaremos el valor `sizeof(TOOLINFO)`.

El campo *uFlags* debe contener algunas banderas que indiquen, por lo menos, el tipo de contenido de otros campos:

- **TTF\_IDISHWND**: indica que el campo *uld* contiene el manipulador de ventana de un control. Si no se especifica, *uld* debe contener un identificador de herramienta, pero esto sólo se aplica a herramientas pertenecientes a barras de herramientas. *uld* también puede ser cero, si usamos un rectángulo para asociarle un tooltip.
- **TTF\_SUBCLASS**: para que el procedimiento de ventana del tooltip intercepte algunos mensajes del ratón, esto nos evita tener que usar el mensaje [TTM\\_RELAYEVENT](#). Normalmente no será necesario controlar los mensajes de ratón para los tooltips, de modo que es más sencillo dejar que ellos los traten internamente.

El resto de las banderas no tiene uso con controles y rectángulos, y los veremos en otros capítulos.

El campo *hwnd* debe contener el manipulador de la ventana padre del tooltip.

*uld* debe contener el manipulador de ventana del control que queremos añadir al tooltip, o cero, si estamos añadiendo un área rectangular.

El campo *rect* indica un área rectangular a la que se asocia el tooltip. Para que se use este rectángulo *uld* debe ser cero, y no debe especificarse la bandera `TTF_IDISHWND`. Si *uld* contiene un identificador o un manipulador de ventana, este campo debe ser cero.

El campo *hinst* se usará sólo si el campo *lpstrText* contiene un identificador de recurso de cadena.

El campo *lpstrText* puede contener una cadena o un identificador de recurso de cadena. También puede tomar el valor `LPSTR_TEXTCALLBACK`, para indicar que la ventana padre debe suministrar una cadena cuando reciba el mensaje de notificación `TTN_GETDISPINFO`. Esta cadena es el texto que se mostrará en el tooltip.

El campo *lParam* sólo se usa con herramientas de barras de herramientas.

El campo *lpReserved* no se usa.

## Asignar tooltip a un control

Para añadir un tooltip a un control (o más propiamente, un control a un tooltip, puesto que el mismo tooltip atiende a varios controles), tenemos que iniciar los miembros *cbSize*, *hwnd*, *uFlags*, *uld* y *lpstrText* de una estructura `TOOLINFO` con los valores adecuados, y enviar el mensaje `TTM_ADDTOOL`.

```
TOOLINFO toolInfo = { 0 };
...

toolInfo.cbSize = sizeof(toolInfo);
toolInfo.hwnd = hwnd;
toolInfo.uFlags = TTF_IDISHWND | TTF_SUBCLASS;
toolInfo.uId = (UINT_PTR)GetDlgItem(hwnd,
```

```
CM_PRUEBA);  
    toolInfo.lpszText = "Este botón emite un 'beep'. ";  
    SendMessage(hwndTip, TTM_ADDTOOL, 0,  
(LPARAM)&toolInfo);
```

## Asignar tooltip a un rectángulo

Si se trata de añadir un área rectangular, además hay que iniciar el miembro *rect*, y dejar *uld* a cero:

```
TOOLINFO toolInfo = { 0 };  
...  
  
    toolInfo.cbSize = sizeof(toolInfo);  
    toolInfo.uFlags = TTF_SUBCLASS;  
    toolInfo.hwnd = hwnd;  
    GetClientRect(hwnd, &toolInfo.rect);  
    toolInfo.lpszText = "Esta es la ventana del ejemplo  
87";  
    SendMessage(hwndTip, TTM_ADDTOOL, 0,  
(LPARAM)&toolInfo);
```

En este ejemplo hemos iniciado el miembro *rect* con el rectángulo que define el área de cliente de la ventana padre. Si la ventana cambia de tamaño, el rectángulo asociado no lo hará, por lo que tendremos que procesar el mensaje [WM\\_SIZE](#) si queremos adaptar el área del tooltip al nuevo tamaño.

## Eliminar un control de un tooltip

También hay que usar una estructura [TOOLINFO](#) en la que inicialemos los miembros *cbSize*, *hwnd* y *uld*. A continuación enviaremos el mensaje [TTM\\_DELTOOL](#), con un puntero a la estructura [TOOLINFO](#).

```
TOOLINFO toolInfo = { 0 };
```

```

...

    toolInfo.cbSize = sizeof(toolInfo);
    toolInfo.hwnd = hwnd;
    toolInfo.uId = (UINT_PTR)GetDlgItem(hwnd,
CM_PRUEBA);
    SendMessage(hwndTip, TTM_DELTOOL, 0,
(LPARAM)&toolInfo);

```

Si se trata de un área rectangular usaremos los campos *cbSize*, *hwnd* y *rect*:

```

TOOLINFO toolInfo = { 0 };
...
    toolInfo.cbSize = sizeof(toolInfo);
    toolInfo.hwnd = hwnd;
    GetClientRect(hwnd, &toolInfo.rect);
    SendMessage(hwndTip, TTM_DELTOOL, 0,
(LPARAM)&toolInfo);

```

Si usamos el mensaje [WM\\_SIZE](#) para adaptar el área del tooltip, deberemos borrarla primero, usando este mensaje y luego volver a crearla.

## Usar cadenas de recursos

Cuando añadimos herramientas a un control tooltip podemos especificar cadenas literales, como en los ejemplos anteriores, pero también podemos usar cadenas procedentes de recursos [STRINGTABLE](#). Para ello basta con usar el identificador del recurso de cadena en el campo *lpszText* de la estructura [TOOLINFO](#), y asignar al campo *hinst* de esa estructura un manipulador de la instancia que contiene el recurso.

Usaremos un fichero de recursos para definir las cadenas:

```

STRINGTABLE

```



```
{  
    STR_HABILITAR "Este botón habilita los tooltips."  
}
```

E inicializaremos los campos de la estructura **TOOLINFO** de la forma adecuada:

```
TOOLINFO toolInfo = { 0 };  
  
toolInfo.cbSize = sizeof(toolInfo);  
toolInfo.hwnd = hwnd;  
toolInfo.uFlags = TTF_IDISHWND | TTF_SUBCLASS;  
toolInfo.uId = (UINT_PTR)GetDlgItem(hwnd,  
CM_HABILITAR);  
toolInfo.hinst = hInstance;  
toolInfo.lpszText = STR_HABILITAR;  
SendMessage(hwndTip, TTM_ADDTOOL, 0,  
(LPARAM)&toolInfo);
```

## Ejemplo 87

### Notificaciones

Los controles tooltip disponen de algunos mensajes de notificación que nos permiten controlar de forma detallada algunos aspectos, veremos ahora algunos de ellos.

Estos mensajes de notificación vienen en el formato de un mensaje **WM\_NOTIFY**, donde **wParam** contiene el identificador del control que envía el mensaje, y **lParam** es un puntero a una estructura **NMHDR**, o una estructura cuyo primer campo es una estructura **NMHDR** y otros campos adicionales, que dependen del control que envía el mensaje. Este es el caso de los mensajes de notificación de los tooltips.

Generalmente, el valor de **wParam** no resulta útil, ya que varios controles pueden tener el mismo identificador. En su lugar se usará

alguno de los campos de [NMHDR](#), o de la estructura específica usada por el control.

## Mensaje de petición de texto

### Nota:

Hay dos mensajes de notificación equivalentes, [TTN\\_GETDISPINFO](#) y [TTN\\_NEEDTEXT](#). Los dos tienen el mismo valor, pero el primero sustituye al segundo, que probablemente quedará obsoleto en el futuro. Cada mensaje tiene una estructura de datos asociada, el primero [NMTTDISPINFO](#) y el segundo [TOOLTIPTTEXT](#), de nuevo, el segundo ha sido sustituido, y no lo usaremos.

El funcionamiento de este mensaje es un poco complicado de explicar, veremos si puedo hacerlo claramente.

Existe una tercera forma de asignar textos a un tooltip. Si cuando añadimos un control a un tooltip especificamos la constante *LPSTR\_TEXTCALLBACK* para el campo *lpstrText*, estaremos indicando que el texto del tooltip para ese control se obtendrá de la propia aplicación cada vez que sea necesario.

```
TOOLINFO toolInfo = { 0 };

    toolInfo.cbSize = sizeof(toolInfo);
    toolInfo.hwnd = hwnd;
    toolInfo.uFlags = TTF_IDISHWND | TTF_SUBCLASS;
    toolInfo.uId = (UINT_PTR)GetDlgItem(hwnd,
CM_DESHABILITAR);
    toolInfo.lpstrText = LPSTR_TEXTCALLBACK; // <-
Asignar texto mediante notificación
    SendMessage(hwndTip, TTM_ADDTOOL, 0,
(LPARAM)&toolInfo);
```

Cuando se vaya a mostrar un tooltip para un control para el que se ha especificado la constante `LPSTR_TEXTCALLBACK` para el campo *lpzText*, se envía un mensaje de notificación `TTN_GETDISPINFO`, a través de un mensaje `WM_NOTIFY` a la ventana padre del tooltip, que deberemos procesar. Cuando recibamos un mensaje de notificación `WM_NOTIFY`, generalmente podremos ignorar el parámetro `wParam`, y en un primer lugar, tomaremos el parámetro `lParam` como un puntero a una estructura `NMHDR`, ya que no sabemos qué tipo de control ha enviado la notificación, y por lo tanto, no sabemos qué estructura viene apuntada por ese parámetro.

```
LPNMHDR pnmhdr;
...
    case WM_NOTIFY:
        pnmhdr = (LPNMHDR)lParam;
        switch(pnmhdr->code) {
...

```

Una vez que sabemos qué tipo de mensaje de notificación hemos recibido, también podremos determinar qué tipo de estructura viene apuntada por `lParam`, y, en el caso concreto del mensaje de notificación `TTN_GETDISPINFO`, sabremos que se trata de una estructura `TOOLTIPTEXT`. Por lo tanto, trataremos a ese parámetro como un puntero a una de esas estructuras.

```
LPNMHDR pnmhdr;
LPNMTTDISPINFO pnmttpdispinfo;
...
    case WM_NOTIFY:
        pnmhdr = (LPNMHDR)lParam;
        switch(pnmhdr->code) {
            case TTN_GETDISPINFO:
                pnmttpdispinfo = (LPNMTTDISPINFO)pnmhdr;

```

A continuación tendremos que identificar el control concreto para el que se ha generado la notificación del tooltip. Para ello tendremos que consultar el miembro *uFlags* de la estructura [NMTTDISPINFO](#), para ver si contiene la bandera TTF\_IDISHWND. En ese caso, el campo *idFrom* de la estructura [NMHDR](#) que es el primer campo de [NMTTDISPINFO](#) contendrá un manipulador de la ventana. Esto será lo normal, ya que nosotros habremos especificado esa bandera al añadir el control al tooltip, pero no está de más verificarlo.

Como ese campo es un manipulador de ventana, podremos obtener el identificador del control mediante una llamada a la función [GetDlgCtrlID](#):

```
LPNMHDR pnmhdr;  
LPNMTTDISPINFO pnmttpdispinfo;  
...  
    case WM_NOTIFY:  
        pnmhdr = (LPNMHDR)lParam;  
        switch(pnmhdr->code) {  
            case TTN_GETDISPINFO:  
                pnmttpdispinfo = (LPNMTTDISPINFO)pnmhdr;  
                if (pnmttpdispinfo->uFlags & TTF_IDISHWND) {  
                    switch(GetDlgCtrlID((HWND)pnmttpdispinfo-  
>hdr.idFrom)) {  
                        case CM_DESHABILITAR:  
                            ...
```

Ahora tenemos varias formas de indicar el texto para el tooltip, elegiremos la que más nos convenga:

- Copiar una cadena de 80 caracteres o menos (incluido en nulo terminador), en el campo *szText* de la estructura [NMTTDISPINFO](#).

```
strcpy(pnmttpdispinfo->szText, "Deshabilita los  
tooltips.");
```



```

"Deshabilita los tooltips.");
        break;
        case CM_HABILITAR:
            pnmtdispinfo->uFlags |=
TTF_DI_SETITEM;
            pnmtdispinfo->lpszText =
"Habilita los tooltips.";
            break;
        }
    }
    break;
}
break;

```

## Ejemplo 88

### Notificaciones de mostrar y ocultar

Cada vez que una ventana tooltip vaya a ser mostrada se envía un mensaje de notificación [TTN\\_SHOW](#) y cuando va a ser ocultada, uno [TTN\\_POP](#). Ambos se envían mediante un mensaje [WM\\_NOTIFY](#), y en ambos casos, el parámetro lParam contiene un puntero a una estructura [NMHDR](#).

No he encontrado ninguna utilidad a estos mensajes. El único ejemplo que se me ha ocurrido es contar cada vez que se muestra u oculta el tooltip, y mostrar esa cuenta en la ventana. También se podría usar esta notificación para visualizar un texto en la barra de estado.

```

case WM_NOTIFY:
    pnmhdr = (LPNMHDR)lParam;
    switch(pnmhdr->code) {
        case TTN_SHOW:
            mostrado++;
            InvalidateRect(hwnd, NULL, TRUE);
            break;
        case TTN_POP:
            ocultado++;
    }
}

```

```

        InvalidateRect(hwnd, NULL, TRUE);
        break;
    }
    break;

...
case WM_PAINT:
    hdc = BeginPaint(hwnd, &ps);
    SetBkMode(hdc, TRANSPARENT);
    sprintf(cad, "Mostrado=%d Ocultado=%d",
mostrado, ocultado);
    TextOut(hdc, 10, 40, cad, strlen(cad));
    EndPaint(hwnd, &ps);
    break;

```

## Personalización

Es posible personalizar un tooltip, cambiando la fuente, los colores o encargando a la aplicación el dibujo de algunos detalles del tooltip.

El mensaje de notificación [NM\\_CUSTOMDRAW](#) se envía para muchos controles, incluídos algunos que ya hemos visto, como por ejemplo, los botones.

En el caso de los tooltips, los cambios de fuente que hagamos sólo tendrán efecto en el estilo clásico. Si activamos los estilos visuales, los cambios de fuente serán ignorados.

En el caso que nos ocupa recibiremos un mensaje de notificación [NM\\_CUSTOMDRAW](#) justo antes de que se empiece a pintar el control, con el código de etapa de dibujo `CDDS_PREPAINT`. Esto nos permite tomar varias acciones:

- Modificar la fuente, en cuyo caso debemos retornar el valor `CDRF_NEWFONT`. Esto indica al procedimiento de ventana del control que debe recalcular el tamaño del texto, y por lo tanto, el del control.
- Modificar el color del texto, mediante la función [SetTextColor](#). En ese caso podemos retornar `CDRF_DODEFAULT`, para que el procedimiento siga su curso normal.

- Retornar el valor `CDRF_NOTIFYPOSTPAINT`, para que volvamos a recibir un mensaje de notificación cuando el tooltip haya sido dibujado del todo. En este caso, el código de etapa será `CDDS_POSTPAINT`.

El resto de los códigos de retorno no se aplican a los tooltips, o al menos no parece que funcionen. En muchos casos porque los tooltips no tienen ítems ni subítems.

Si hemos retornado `CDRF_NOTIFYPOSTPAINT` recibiremos un mensaje de notificación con el código `CDDS_POSTPAINT`. Podemos aprovechar esta notificación para hacer retoques en el tooltip, y digo retoques, porque el tooltip ya estará pintado. Por supuesto, esos "retoques" pueden incluir borrar el contenido completo del tooltip y volver a pintarlo.

En este ejemplo procesamos los mensaje `WM_NOTIFY` para que el tooltip del control identificado con `CM_BOTON3` se muestre con otra fuente, en color rojo y con un borde grueso.

```
LPNMHDR pnmhdr;
LPNMTTCUSTOMDRAW pnmttpcustomdraw;

...
    case WM_CREATE:
...
        hFont = CreateFont(-30, 0, 0, 450, 300, FALSE,
FALSE, FALSE,
                        DEFAULT_CHARSET, OUT_TT_PRECIS,
CLIP_DEFAULT_PRECIS,
                        PROOF_QUALITY, DEFAULT_PITCH | FF_ROMAN,
"Times New Roman");
        hPen = CreatePen(PS_SOLID, 4, RGB(255,0,0));
...
    case WM_NOTIFY:
        pnmhdr = (LPNMHDR)lParam;
        switch(pnmhdr->code) {
            case TTN_SHOW:
                mostrado++;
                InvalidateRect(hwnd, NULL, TRUE);
                break;
            case TTN_POP:
                ocultado++;
```



```

        InvalidateRect(hwnd, NULL, TRUE);
        break;
    case NM_CUSTOMDRAW:
        pnmtdcustomdraw = (LPNMTCUSTOMDRAW)pnmhdr;
        if(pnmtdcustomdraw->nmcdr.hdr.hwndFrom ==
hwndTip &&
            GetDlgCtrlID((HWND)pnmtdcustomdraw-
>nmcdr.hdr.idFrom) == CM_BOTON3 &&
            pnmtdcustomdraw->nmcdr.dwDrawStage ==
CDDS_PREPAINT) {
            SelectObject(pnmtdcustomdraw->nmcdr.hdc,
hFont);
            SetTextColor(pnmtdcustomdraw->nmcdr.hdc,
RGB(255,0,0));
            return CDRF_NEWFONT |
CDRF_NOTIFYPOSTPAINT;
        } else
            if(pnmtdcustomdraw->nmcdr.hdr.hwndFrom ==
hwndTip &&
            GetDlgCtrlID((HWND)pnmtdcustomdraw-
>nmcdr.hdr.idFrom) == CM_BOTON3 &&
            pnmtdcustomdraw->nmcdr.dwDrawStage ==
CDDS_POSTPAINT) {
            SelectObject(pnmtdcustomdraw->nmcdr.hdc,
GetObject(HOLLOW_BRUSH));
            SelectObject(pnmtdcustomdraw->nmcdr.hdc,
hPen);
            Rectangle(pnmtdcustomdraw->nmcdr.hdc,
pnmtdcustomdraw->nmcdr.rc.left,
pnmtdcustomdraw->nmcdr.rc.top,
pnmtdcustomdraw-
>nmcdr.rc.right,
pnmtdcustomdraw-
>nmcdr.rc.bottom);
            return CDRF_SKIPDEFAULT;
        } else
            return CDRF_DODEFAULT;
        break;
    }
    break;
...
case WM_DESTROY:
    DeleteObject(hFont);
    DeleteObject(hPen);
...

```

## Ejemplo 89

### Otros mensajes

Los tooltips disponen de muchos otros mensajes, algunos de los cuales los explicaremos cuando veamos otros controles comunes, como tree-views y list-views, y otros que probablemente no veamos, porque tienen poca utilidad.

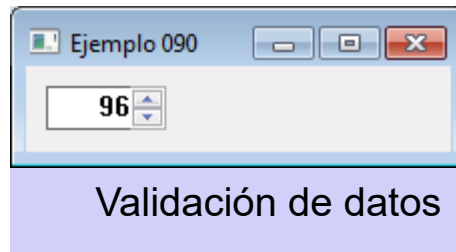
Esta es una lista del resto de los mensajes que no hemos visto en este capítulo:

- TTM\_ADJUSTRECT
- TTM\_ENUMTOOLS
- TTM\_GETBUBBLESIZE
- TTM\_GETCURRENTTOOL
- TTM\_GETDELAYTIME
- TTM\_GETMARGIN
- TTM\_GETMAXTIPWIDTH
- TTM\_GETTEXT
- TTM\_GETTIPBKCOLOR
- TTM\_GETTIPTEXTCOLOR
- TTM\_GETTITLE
- TTM\_GETTOOLCOUNT
- TTM\_GETTOOLINFO
- TTM\_HITTEST
- TTM\_NEWTOOLRECT
- TTM\_POP
- TTM\_POPUP
- TTM\_RELAYEVENT
- TTM\_SETDELAYTIME
- TTM\_SETMARGIN
- TTM SETTIPBKCOLOR
- TTM SETTIPTEXTCOLOR
- TTM SETTOOLINFO
- TTM\_SETWINDOWTHEME
- TTM\_TRACKACTIVATE

- TTM\_TRACKPOSITION
- TTM\_UPDATE
- TTM\_UPDATETIPTTEXT
- TTM\_WINDOWFROMPOINT

# Capítulo 52 Control UpDown

Un control UpDown está formado por un par de botones con flechas. Esos botones se usan para incrementar o decrementar un valor, generalmente asociado a otro control. Cuando este control asociado existe, se le denomina ventana amiga (buddy window). El valor asociado se denomina posición actual.



Desde el punto de vista del usuario, el control UpDown y su ventana amiga se comportan como un único control, de modo que las flechas constituyen una forma alternativa de modificar el valor mostrado por el control.

## Creación de un control UpDown

Para crear un control UpDown se puede usar la función [CreateUpDownControl](#), aunque no es muy aconsejable, ya que los valores de la posición y rango estarán limitados a 16 bits. De hecho, esta función se considera obsoleta.

En su lugar es preferible usar la función [CreateWindowEx](#), especificando el valor `UPDOWN_CLASS` como clase de ventana de control. Es imprescindible indicar al menos los estilos `WS_CHILD` y `WS_VISIBLE`, además de los específicos para estos controles que consideremos necesarios en nuestro caso.

```
CreateWindowEx(0, UPDOWN_CLASS, NULL,
               WS_CHILD | WS_VISIBLE | UDS_ALIGNRIGHT |
               UDS_SETBUDDYINT | UDS_WRAP,
               10, 10,
               20, 20,
```

```
hwnd, (HMENU)100,  
hInstance, NULL);
```

Como con el resto de los controles comunes, es necesario asegurarse de que la DLL ha sido cargada mediante una llamada a la función [InitCommonControlsEx](#) indicando el valor de bandera `ICC_UPDOWN_CLASS` en el miembro `dwICC` de la estructura [INITCOMMONCONTROLSEX](#) que pasaremos como parámetro.

```
INITCOMMONCONTROLSEX icce;  
...  
icce.dwSize = sizeof(INITCOMMONCONTROLSEX);  
icce.dwICC = ICC_UPDOWN_CLASS;  
InitCommonControlsEx(&icce);
```

La apariencia de este tipo de controles también se ve afectada si se activan o no los estilos visuales. La apariencia con los estilos visuales activos es algo mejor, ya que el conjunto del control UpDown y la ventana amiga tienen una apariencia más homogénea y la idea de que se trata de un control único es más evidente.

Decordemos que para activar los estilos visuales hay que incluir un fichero de manifiesto en el fichero de recursos.

## Especificar una ventana amiga

Generalmente no crearemos controles de este tipo sin una ventana asociada, de modo que tendremos que asociarle una.

Hay dos formas de hacer esto:

- Automáticamente: si indicamos el estilo [UDS\\_AUTOBUDDY](#), se tomará como ventana amiga para el control la anterior en el orden Z, que será el control que hayamos creado justo antes.

```
CreateWindowEx(0, "EDIT", "100", ES_RIGHT |
```

```

ES_NUMBER | WS_VISIBLE | WS_CHILD | WS_BORDER,
                                10, 10, 60, 22, hwnd, (HMENU)100,
hInstance, NULL);
    CreateWindowEx(0, UPDOWN_CLASS, NULL,
                                WS_CHILD | WS_VISIBLE |
UDS_ALIGNRIGHT | UDS_WRAP | UDS_SETBUDDYINT |
UDS_AUTOBUDDY,
                                0, 0,
                                0, 0,
                                hwnd, (HMENU)101,
                                hInstance, NULL);

```

- Manualmente: enviando un mensaje [UDM\\_SETBUDDY](#).

```

    CreateWindowEx(0, "EDIT", "100", ES_RIGHT |
ES_NUMBER | WS_VISIBLE | WS_CHILD | WS_BORDER,
                                10, 10, 60, 22, hwnd, (HMENU)100,
hInstance, NULL);
    CreateWindowEx(0, UPDOWN_CLASS, NULL,
                                WS_CHILD | WS_VISIBLE |
UDS_ALIGNRIGHT | UDS_WRAP | UDS_SETBUDDYINT,
                                0, 0,
                                0, 0,
                                hwnd, (HMENU)101,
                                hInstance, NULL);
    SendDlgItemMessage(hwnd, 101, UDM_SETBUDDY,
(WPARAM)GetDlgItem(hwnd, 100), 0);

```

**Nota:** La asociación más frecuente es entre un control UpDown y un control edit. Este conjunto forma lo que normalmente se conoce como un control spinner (control giratorio). Sin embargo, en muchos sitios se conocen los controles UpDown como Controles Spin, por ejemplo, en el editor de recursos "ResEdit".

De forma simétrica, para obtener el manipulador de ventana de la ventana amiga asociada a un control UpDown, se usa el mensaje [UDM\\_GETBUDDY](#).

## Estilos

Hay varios estilos que se pueden aplicar a los controles UpDown, además de UDS\_AUTOBUDDY.

UDS\_ALIGNLEFT y UDS\_ALIGNRIGHT afectan a la posición del control con respecto a la ventana amiga asociada. El primer estilo alinea el control a la izquierda y el segundo a la derecha de la ventana amiga.

Si se especifica UDS\_HORZ, las flechas del control apuntan a derecha e izquierda, si se omite, apuntan arriba y abajo.

Si se especifica el estilo UDS\_HOTTRACK, se modificará el aspecto del control cuando el cursor del ratón está sobre él, resaltando la flecha correspondiente.

El estilo UDS\_SETBUDDYINT provoca que se modifique el texto asociado a la ventana amiga, enviando un mensaje [WM\\_SETTEXT](#) con el valor actual de control UpDown.

El estilo UDS\_NOTHOUSANDS omite los puntos separadores de miles en el texto asociado a la ventana amiga. Si no se especifica, se insertarán puntos separadores.

UDS\_ARROWKEYS hace que las flechas del cursor se comporten como si se pulsarán las flechas del control.

Por último UDS\_WRAP ajusta los valores del control dentro de los márgenes indicados, de modo que si se sobrepasa el máximo, se vuelve al valor mínimo, y si se baja por debajo del mínimo, se vuelve al máximo. Si no se indica este estilo, al llegar al máximo se mantiene el valor, y sólo es posible disminuirlo, y al llegar al mínimo, sólo se puede incrementar el valor actual.

## Rango y posición actual

Del mismo modo que los controles de desplazamiento, podemos fijar los límites inferior y superior de los controles UpDown, de modo que la posición actual no pueda tomar valores fuera de ese rango.

Para hacerlo disponemos de dos mensajes [UDM\\_SETRANGE](#) y [UDM\\_SETTANGE32](#). El primero para rangos especificados por valores de 16 bits, y el segundo para valores de 32 bits.

```
SendDlgItemMessage(hwnd, 101, UDM_SETRANGE32,  
(WPARAM)-100, (LPARAM)100);  
SendDlgItemMessage(hwnd, 101, UDM_SETRANGE, 0,  
(LPARAM) MAKELONG((short) 100, (short) -100));
```

Para modificar el valor actual de la posición del control también disponemos de dos mensajes, [UDM\\_SETPOS](#) y [UDM\\_SETPOS32](#), el primero cuando el rango esté definido con valores de 16 bits, y el segundo para rangos de 32 bits.

Los mensajes complementarios sirven para leer los valores de rangos y posiciones actuales:

El mensaje [UDM\\_GETRANGE](#) obtiene el rango empaquetado en un entero de 32 bits. La palabra de menor peso contiene el límite superior, y la de mayor peso, el inferior.

El mensaje [UDM\\_GETRANGE32](#) obtiene el rango en forma de enteros de 32 bits. Para ello hay que pasarle dos parámetros. En wParam un puntero a un entero con signo que recibirá el límite inferior, y en lParam un puntero a un entero con signo que recibirá el límite superior. Cualquiera de los punteros puede ser NULL, si no queremos obtener alguno de los límites.

Por último, para obtener la posición actual se usa el mensaje [UDM\\_GETPOS](#) o [UDM\\_GETPOS32](#). El primero con una precisión de 16 bits, y el segundo con 32 bits de precisión.

## Ficheros de recursos

Para usar controles UpDown a un cuadro de diálogo dentro de un fichero de recursos basta con añadir un control de la clase *UPDOWN\_CLASS*. Si además se especifica el estilo *UDS\_AUTOBUDDY*, hay que tener especial cuidado en el orden en que se definen los controles, ya que el control UpDown se asociará con el control especificado inmediatamente antes:



```
EDITTEXT          ID_EDIT5, 10, 9, 93, 14, ES_AUTOHSCROLL
CONTROL           "", ID_UPDOWN5, UPDOWN_CLASS,
UDS_ARROWKEYS | UDS_AUTOBUDDY | UDS_HOTTRACK |
UDS_SETBUDDYINT, 104, 9, 11, 14
```

## Aceleradores

Es probable que ya lo hayas notado, pero los controles UpDown no se comportan del mismo modo si se pulsa una vez sobre una de sus flechas que si se mantiene pulsado durante un tiempo. Cuanto más tiempo se mantiene pulsada una de las flechas, a mayor velocidad cambia su valor actual. Este comportamiento es lo que se llama "aceleradores", y se puede modificar a nuestra discreción para adaptarlo a nuestro gusto o a nuestras necesidades.

Los aceleradores trabajan por tramos, y podemos añadir tantos de esos tramos como queramos. Cada uno de ellos se define por dos valores, el primero es un tiempo expresado en segundos, y el segundo es el valor del incremento que se aplica a partir de que transcurra ese tiempo.

Para cada tramo se usa una estructura [UDACCEL](#). Si queremos usar aceleradores en varios tramos, crearemos un array de estas estructuras.

Supongamos que queremos que el primer tramo efectúe incrementos de uno en uno, el segundo de diez en diez, a partir de que transcurran cinco segundos de pulsación, el tercero de cincuenta en cincuenta, a partir de los diez segundos de pulsación (contando desde el principio). El cuarto de cien en cien, a partir de los quince segundos. El array quedaría de esta forma:

```
UDACCEL uda[] = {
    {0, 1},
    {5, 10},
    {10, 50},
```

```
        {15, 100}  
    };
```

Es importante tener en cuenta que los tiempos se cuentan siempre desde el principio de la pulsación, de modo que la estructura anterior cambia los incrementos cada cinco segundos.

También hay que tener presente que los valores actuales del control no serán exactamente el resultado de aplicar el incremento sobre el valor actual, sino que los valores actuales serán múltiplos del incremento. Por ejemplo, si partimos de cero, y aplicamos la tabla de aceleradores anteriores, durante los primeros cinco segundos se aplican incrementos de una unidad. Los incrementos no son cada segundo, y desde el primero al segundo incremento pasa más tiempo que en los sucesivos, de modo que no podemos prever cual será el valor actual del cursor al cabo de cinco segundos, pero supongamos que es 42. A partir de ese momento, los incrementos serán múltiplos de diez, de modo que el siguiente valor no será 52, sino 50, y a partir de ahí, y durante los siguientes cinco segundos, los incrementos serán de diez en diez. Y así sucesivamente.

Cada vez que se suelte el botón del ratón, la tabla de aceleradores volverá al comienzo.

Para asignar una tabla de aceleradores a un control se usa el mensaje [UDM\\_SETACCEL](#), indicando como parámetro wParam el número de tramos, y en lParam el puntero al primer elemento del array de tramos:

```
SendDlgItemMessage(hwnd, ID_UPDOWN2, UDM_SETACCEL,  
(WPARAM) sizeof(uda) / sizeof(UDACCEL), (LPARAM) uda);
```

Para recuperar la tabla de aceleradores de un control podemos usar el mensaje [UDM\\_GETACCEL](#). En wParam indicaremos el número máximo de tramos que podemos recuperar, dependiendo del tamaño del array en el que los almacenaremos. En lParam

indicaremos un puntero al primer elemento del array que recibirá la tabla de aceleradores. El valor de retorno será el número de tramos recuperados:

```
int i;  
i=SendDlgItemMessage(hwnd, ID_UPDOWN3, UDM_GETACCEL,  
4, (LPARAM)uda);
```

## Bases de numeración

Cuando se usa el estilo UDS\_SETBUDDYINT, el texto de la ventana amiga asociada al control se actualiza automáticamente según el valor actual del control. En este caso, tenemos dos posibilidades a la hora de elegir la base de numeración para expresar esos valores: decimal o hexadecimal. Para establecer el valor de la base de numeración a utilizar podemos usar el mensaje [UDM\\_SETBASE](#), indicando en wParam el valor de la base deseada, que sólo puede ser 10 ó 16.

```
SendDlgItemMessage(hwnd, ID_UPDOWN1, UDM_SETBASE,  
16, 0);
```

Para recuperar el valor actual de la base de numeración utilizada por un control, se usa el mensaje [UDM\\_GETBASE](#), con los dos parámetros a cero. El valor de la base se obtiene en el valor de retorno.

```
i = SendDlgItemMessage(hwnd, ID_UPDOWN1,  
UDM_GETBASE, 0, 0);
```

## Mensajes de notificación

Los controles UpDown pueden enviar tres mensajes diferentes a sus ventanas padre. Dos de ellos dependen del estilo, si es UDS\_HORZ enviarán mensajes [WM\\_HSCROLL](#) cada vez que se pulse una de las flechas. Si no se especifica ese estilo, el control tendrá orientación vertical, y en su lugar enviará mensajes [WM\\_VSCROLL](#).

Podemos usar estos mensajes para actualizar la ventana amiga asociada de forma manual, de modo que podamos personalizar el valor mostrado. De ese modo podemos añadir caracteres de formato, o trabajar con valores no enteros, o incluso con valores no numéricos, siempre y cuando a cada posible valor del ámbito de valores del control UpDown le hagamos corresponder un valor de otro dominio.

Por ejemplo, podemos asociar un control edit de sólo lectura a un control UpDown para elegir frutas dentro de los valores definidos en un array de cadenas:

```
const char *fruta[] = {
    "Peras",
    "Manzanas",
    "Naranjas",
    "Melocotones",
    "Plátanos",
    "Fresas",
    "Castañas",
    "Nueces",
    "Piñas"
};

...
case WM_CREATE:
    CreateWindowEx(0, "EDIT", "", ES_READONLY |
WS_VISIBLE | WS_CHILD | WS_BORDER,
                    10, 40, 120, 22, hwnd,
    (HMENU)ID_EDIT4, hInstance, NULL);
    CreateWindowEx(0, UPDOWN_CLASS, NULL,
WS_CHILD | WS_VISIBLE |
UDS_ALIGNRIGHT | UDS_ARROWKEYS | UDS_AUTOBUDDY |
UDS_HOTTRACK,
                    0, 0,
                    0, 0,
```

```

                                hwnd, (HMENU)ID_UPDOWN4,
                                hInstance, NULL);
        SendDlgItemMessage(hwnd, ID_UPDOWN4,
UDM_SETRANGE32, (WPARAM)0, (LPARAM)
        (sizeof(fruta)/sizeof(fruta[0])-1));
        SendDlgItemMessage(hwnd, ID_UPDOWN4,
UDM_SETPOS32, 0, (LPARAM)0);
        SendDlgItemMessage(hwnd, ID_EDIT4, WM_SETTEXT,
0, (LPARAM)fruta[0]);
        ...
        case WM_VSCROLL:
            if((HWND)lParam == GetDlgItem(hwnd, ID_UPDOWN4))
            {
                SendDlgItemMessage(hwnd, ID_EDIT4,
WM_SETTEXT, 0, (LPARAM)fruta[SendDlgItemMessage(hwnd,
ID_UPDOWN4, UDM_GETPOS32, 0, 0)]);
            }
            break;

```

Nada nos impide asociar el control UpDown con un control estático para elegir opciones mostrando iconos o mapas de bits.

El tercer mensaje es un mensaje de notificación, que se recibe a través de un mensaje [WM\\_NOTIFY](#). Este mensaje se envía cada vez que la posición actual del control UpDown vaya a ser modificada. Al procesar este mensaje podemos permitir o evitar que tal modificación tenga efecto.

Podemos usar esta posibilidad para hacer que los rangos de los valores del control UpDown varíen de forma dinámica, por ejemplo, en función de los valores de otros controles, o para restringir los valores dentro del rango definido por [UDM\\_SETRANGE](#) o [UDM\\_SETRANGE32](#), etc.

Al recibir el mensaje de notificación, [WM\\_NOTIFY](#), en lParam recibiremos un puntero a una estructura [NMUPDOWN](#).

Esta estructura contiene en primer lugar una estructura [NMHDR](#), que es una cabecera común a todos los mensajes de notificación que contiene un manipulador de ventana del control que envía el mensaje, el identificador del control que envía el mensaje y un código de notificación que especifica el motivo.

Usaremos el código para saber de qué mensaje de notificación se trata, y el identificador de control para saber qué control lo envía.

Los otros dos campos de la estructura `NMUPDOWN`, *iDelta* e *iPos* contienen, respectivamente, el próximo incremento del valor de la posición, y la posición actual del control UpDown.

Al procesar este mensaje, si retornamos con TRUE, no se tendrá en cuenta el incremento solicitado por el usuario. Si se retorna con FALSE, sí. Además, se tendrá en cuenta cualquier modificación que hagamos de los valores de *iDelta* o *iPos*, de modo que podremos influir en el valor actual del control, independientemente del incremento solicitado por el usuario y del valor previo del control.

Por ejemplo, podemos evitar que el control tome valores que sean múltiplos de tres:

```
NM_UPDOWN *nmup;
LPNMHDR pnmhdr;
...
case WM_NOTIFY:
    pnmhdr = (LPNMHDR)lParam;
    switch(pnmhdr->code) {
        case UDN_DELTAPOS:
            if(pnmhdr->idFrom == ID_UPDOWN3) {
                nmup = (NM_UPDOWN*)lParam;
                if(!((nmup->iPos+nmup->iDelta) % 3))
                    if(nmup->iDelta > 0)
                        nmup->iDelta++; else nmup->iDelta--;
                return FALSE;
            }
            break;
    }
    break;
```

## Ejemplo 90

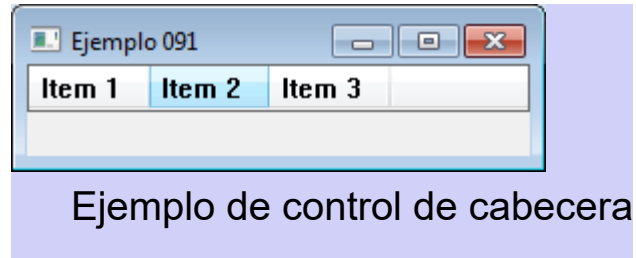
# Capítulo 53 Control de cabecera

Un control de cabecera es una ventana estrecha, en forma de barra, dividida en zonas, a las que denominamos ítems. Generalmente se usan con listas formadas por varias

columnas, a cada una de las cuales le corresponde un ítem. El usuario puede modificar la anchura de cada ítem arrastrando el divisor que se encuentra entre los ítems.

Los controles de cabecera no están pensados para ser usados como controles individuales, sino como controles hijos de otros controles, como listas. Sin embargo, asociar un control de cabecera a un control lista no funcionará como esperaríamos directamente, ya que los controles lista no están preparados para mostrar información en varias columnas. En esos casos deberemos crear una subclase del control lista, y además, la lista deberá ser owner-draw.

Sin embargo, hay otros controles comunes que incluyen un control de cabecera, como el `ListView`, que aún no hemos visto. Lo que aprendamos sobre controles de cabecera nos servirá para aplicarlo a estos controles, ya que podremos conseguir un manipulador al control de cabecera asociado a ellos.



## Creación de un control de cabecera

Crear un control de cabecera requiere varios pasos. Para empezar, usaremos la función [CreateWindowEx](#), especificando el valor `WC_HEADER` como clase de ventana de control. Es

imprescindible indicar al menos el estilo `WS_CHILD`, además de los específicos para estos controles que consideremos necesarios en nuestro caso.

```
CreateWindowEx(0, WC_HEADER, NULL,
               WS_CHILD | HDS_BUTTONS | HDS_HORZ,
               0, 0, 0, 0,
               hwnd, (HMENU)ID_HEADER1,
               hInstance, NULL);
```

No necesitamos especificar una posición ni las dimensiones del control, ya que a continuación las calcularemos en función de la ventana padre y moveremos el control a esa posición.

Para calcular la posición y dimensiones del control usaremos el mensaje `HDM_LAYOUT` o la macro `Header_Layout`, ambos son equivalentes. Antes, iniciaremos los campos *prc* y *pwpos* de una estructura `HDLAYOUT`.

El campo *prc* debe contener un puntero a una estructura `RECT` con las dimensiones del área de cliente de la ventana o control padre del control de cabecera.

El campo *pwpos* debe contener un puntero a una estructura `WINDOWPOS`, que recibirá las coordenadas de la posición y las dimensiones que debe tener el control de cabecera, según el tamaño y posición del rectángulo indicado.

```
RECT rc;
HDLAYOUT hdl;
WINDOWPOS wp;
...
    GetClientRect(hwnd, &rc);
    hdl.prc = &rc;
    hdl.pwpos = &wp;
    SendDlgItemMessage(hwnd, ID_HEADER1, HDM_LAYOUT, 0,
(LPARAM) &hdl);
    /* O bien la macro:
    Header_Layout(GetDlgItem(hwnd, ID_HEADER1), &hdl);
    */
```



Una vez tenemos las dimensiones y coordenadas del control, sólo queda moverlo a esa posición y modificar su tamaño. Para ello usaremos la función [SetWindowPos](#), y aprovecharemos para hacerlo visible, añadiendo la bandera `SWP_SHOWWINDOW`:

```
SetWindowPos(GetDlgItem(hwnd, ID_HEADER1),
wp.hwndInsertAfter, wp.x, wp.y,
wp.cx, wp.cy, wp.flags | SWP_SHOWWINDOW);
```

Como con el resto de los controles comunes, es necesario asegurarse de que la DLL ha sido cargada mediante una llamada a la función [InitCommonControlsEx](#) indicando el valor de bandera `ICC_LISTVIEW_CLASSES` en el miembro `dwICC` de la estructura [INITCOMMONCONTROLSEX](#) que pasaremos como parámetro.

```
INITCOMMONCONTROLSEX icce;
...
icce.dwSize = sizeof(INITCOMMONCONTROLSEX);
icce.dwICC = ICC_LISTVIEW_CLASSES;
InitCommonControlsEx(&icce);
```

La apariencia de este tipo de controles también se ve afectada si se activan o no los estilos visuales.

Recordemos que para activar los estilos visuales hay que incluir un fichero de manifiesto en el fichero de recursos.

## Añadir columnas

Para insertar columnas en un control de cabecera se usa el mensaje [HDM\\_INSERTITEM](#), indicando en el parámetro `wParam` el índice de la columna a continuación de la cual se insertará la nueva, y el `lParam` un puntero a una estructura [HDITEM](#).

En la estructura **HDITEM** el parámetro *mask* indica qué miembros opcionales de la estructura contienen valores válidos.

Cuando se trate de columnas con texto, deberemos indicar los valores de *pszText* y *cchTextMax*, con los valores del texto y su longitud máxima, respectivamente.

*cxy* indica la anchura o altura de la columna o fila, dependiendo de si el control es horizontal o vertical.

También se puede usar un mapa de bits, mediante el miembro *hbm*, o bien una imagen dentro de una lista de imágenes indicando el índice en el miembro *ilImage*.

El miembro *lParam* permite usar un dato asociado al ítem, definido por la aplicación.

*iOrder* sirve para indicar el orden en que se mostrará la columna dentro del control. Esto puede ser útil cuando se quieren conservar las preferencias de orden del usuario, en lugar de crear el control siempre con el orden original por defecto.

El puntero *pvFilter* se usa para asignar un filtro a la columna. El tipo de filtro se indica en el miembro *type*, que puede tomar los valores HDFT\_ISSTRING para cadena, HDFT\_ISNUMBER para valores enteros, HDFT\_ISDATE para fechas y HDFT\_HASNOVALUE para ignorar el filtro.

Cuando se define un tipo concreto el sistema impide que se introduzcan valores no permitidos, por ejemplo, no será posible introducir texto en un filtro de tipo HDFT\_ISNUMBER o HDFT\_ISDATE.

El miembro *state* indica el estado de la columna, el valor HDIS\_FOCUSED indica que la columna tiene el foco del teclado.

El miembro *fmt* sirve para determinar el formato: si se trata de imágenes o texto, y en ese caso, si el texto se muestra centrado, a la derecha, a la izquierda o de derecha a izquierda. Se puede añadir una imagen de flecha arriba o flecha abajo, para indicar el orden en que aparecen los elementos de la columna mediante los valores HDF\_SORTUP o HDF\_SORTDOWN, respectivamente. El valor HDF\_CHECKBOX indica que se debe mostrar un checkbox, y si se

indica además el valor `HDF_CHECKED` se mostrará marcado. `HDF_FIXEDWIDTH` impide que el usuario pueda modificar la anchura de la columna, `HDF_SPLITBUTTON` muestra un botón de despliegue.

```
int InsertarItem(HWND hwndHeader, int iDespuesDe, int
nAncho, LPTSTR lpsz)
{
    HDITEM hdi;
    int index;

    hdi.mask = HDI_TEXT | HDI_FORMAT | HDI_WIDTH;
    hdi.cxy = nAncho;
    hdi.pszText = lpsz;
    hdi.cchTextMax =
strlen(hdi.pszText)/sizeof(hdi.pszText[0]);
    hdi.fmt = HDF_LEFT | HDF_STRING;

    index = SendMessage(hwndHeader, HDM_INSERTITEM,
(WPARAM) iDespuesDe, (LPARAM) &hdi);

    return index;
}
```

## Cambio de tamaño de la ventana padre

Cada vez que el tamaño de la ventana padre del control cambie, deberemos ajustar el tamaño y posición del control. En esos casos recibiremos un mensaje `WM_SIZE`, y deberemos repetir los pasos anteriores:

```
case WM_SIZE:
    GetClientRect(hwnd, &rc);
    hdl.prc = &rc;
    hdl.pwpos = &wp;
    SendDlgItemMessage(hwnd, ID_HEADER1, HDM_LAYOUT, 0,
(LPARAM) &hdl);
    SetWindowPos(GetDlgItem(hwnd, ID_HEADER1),
```

```
wp.hwndInsertAfter, wp.x, wp.y,  
wp.cx, wp.cy, wp.flags | SWP_SHOWWINDOW);
```

## Estilos

Hay varios **estilos** que se pueden aplicar a los controles de cabecera.

**HDS\_BUTTONS**: cuando se requiera que la aplicación realice alguna tarea (como seleccionar o copiar) cuando el usuario pulsa sobre alguna de las columnas, este estilo hace que cada encabezado se comporte como un botón. Si no se usa, cada ítem se comporta como una etiqueta estática.

**HDS\_DRAGDROP**: permite arrastrar los elementos del encabezado.

**HDS\_FILTERBAR**: añade una segunda barra con filtros.

**HDS\_FLAT**: produce una apariencia plana.

**HDS\_FULLDRAG**: para que se siga mostrando el contenido de la columna mientras el usuario cambia su tamaño. Si no se indica sólo se muestra el borde hasta que el usuario establece el nuevo tamaño.

**HDS\_HIDDEN**: Indica un control de encabezado que se va a ocultar. Este estilo no oculta el control. En su lugar, cuando se envía el mensaje de **HDM\_LAYOUT** a un control de encabezado con el estilo de **HDS\_HIDDEN**, el control devuelve cero en el miembro **CY** de la estructura **windowpos** ( ). A continuación, ocultaría el control estableciendo su alto en cero. Esto puede ser útil si desea usar el control como contenedor de información en lugar de como un control visual.

**HDS\_HORZ**: Crea un control de encabezado con una orientación horizontal. Hasta donde he podido ver, esta es la opción por defecto, y no es posible crear controles de encabezado verticales.

**HDS\_HOTTRACK**: Habilita el seguimiento activo. Este estilo no parece tener ninguna funcionalidad en controles de cabecera, pero

sí se usa en controles ListView, que contienen en su composición un control de cabecera.

**HDS\_CHECKBOXES:** Permite colocar cajas de chequeo en los elementos de encabezado.

**HDS\_NOSIZING:** El usuario no puede modificar la anchura de los items arrastrando el divisor.

**HDS\_OVERFLOW:** Se muestra un botón cuando no se pueden mostrar todos los elementos dentro del rectángulo del control de encabezado. Al hacer clic en este botón, se envía una notificación HDN\_OVERFLOWCLICK.

## Mensajes de gestión de columnas

Además del mensaje **HDM\_INSERTITEM** que vimos antes, también disponemos del mensaje **HDM\_GETITEMCOUNT** que obtiene el número de columnas actualmente en el control indicado.

```
int n;  
n = SendDlgItemMessage(hwnd, ID_HEADER1, HDM_GETITEMCOUNT,  
0, 0);
```

El mensaje **HDM\_DELETEITEM** permite eliminar una columna, en wParam se debe indicar el índice de la columna a eliminar.

```
SendDlgItemMessage(hwnd, ID_HEADER1, HDM_DELETEITEM,  
(WPARAM) 2, 0);
```

El mensaje **HDM\_SETITEM** permite modificar una columna existente. En wParam indicaremos el índice de la columna a modificar y en lParam un puntero a una estructura **HDITEM** con los nuevos valores de la columna. Por lo demás funciona igual que **HDM\_INSERTITEM**.

En este ejemplo modificamos el formato de la segunda columna (los índices empiezan en 0):

```
HDITEM hdi;  
hdi.mask = HDI_FORMAT;  
hdi.fmt = HDF_STRING | HDF_CHECKBOX | HDF_CENTER |  
HDF_SORTDOWN | HDF_SPLITBUTTON;  
  
SendDlgItemMessage(hwnd, ID_HEADER1, HDM_SETITEM,  
(WPARAM)1, (LPARAM)&hdi);
```

Por último, el mensaje [HDM\\_GETITEM](#) permite recuperar la información de un ítem de columna. En wParam se indica el índice del ítem a recuperar y en lParam un puntero a una estructura [HD\\_ITEM](#) o [HDITEM](#) en el que se devolverá la información. En el miembro *mask* indicaremos qué valores queremos recuperar.

```
HDITEM hdi;  
hdi.mask = HDI_FILTER; // Recuperar información del filtro  
SendDlgItemMessage(hwnd, ID_HEADER1, HDM_GETITEM,  
(WPARAM)1, (LPARAM)&hdi);
```

## Mensajes relacionados con el orden de columnas

Dado que las columnas se pueden mover usando operaciones de arrastre, o mediante mensajes, el índice de cada una no tiene por qué corresponder con su posición dentro del control de cabecera. Existen varios mensajes para relacionar índices y posiciones de columnas.

El mensaje [HDM\\_GETORDERARRAY](#) obtiene un array de índices de columnas en el orden en que aparecen en el control de izquierda a derecha. En lParam se pasa un puntero a un array de

enteros que recibirán los índices, y en wParam se indica el valor de valores a recuperar, que como máximo será el número de columnas.

```
int *orden;
int n;
n = SendDlgItemMessage(hwnd, ID_HEADER1,
HDM_GETITEMCOUNT, 0, 0);
if((orden = (int*)calloc(n, sizeof(int)))) {
    SendDlgItemMessage(hwnd, ID_HEADER1,
HDM_GETORDERARRAY, (WPARAM)n, (LPARAM)orden);
    free(orden);
}
```

De forma análoga, podemos definir un nuevo orden para las columnas usando el mensaje [HDM\\_SETORDERARRAY](#). Los parámetros tiene el mismo significado que en el mensaje anterior.

```
int *orden;
int n;
n = SendDlgItemMessage(hwnd, ID_HEADER1,
HDM_GETITEMCOUNT, 0, 0);
if((orden = (int*)calloc(n, sizeof(int)))) {
    orden[0] = 2;
    orden[1] = 1;
    orden[2] = 0;
    SendDlgItemMessage(hwnd, ID_HEADER1,
HDM_SETORDERARRAY, (WPARAM)n, (LPARAM)orden);
    free(orden);
}
```

El mensaje [HDM\\_ORDERTOINDEX](#) obtiene el índice de una columna a partir de su posición de izquierda a derecha. En wParam se indica el valor del orden, y lParam no se usa.

```
int i = SendDlgItemMessage(hwnd, ID_HEADER1,
HDM_ORDERTOINDEX, (WPARAM)1, 0);
```

El mensaje `HDM_HITTEST` sirve para verificar si un punto determinado de la pantalla pertenece a un control de cabecera o a un ítem concreto de un control de cabecera, y en ese caso a cual.

El punto se proporciona en el miembro *pt*, de tipo `POINT` dentro de una estructura `HDHITTESTINFO` cuya dirección se pasa en el parámetro `lParam`. Se retorna el índice de la columna que contiene el punto, o -1 si no pertenece a ninguna. También se retornan el resto de los miembros de la estructura `HDHITTESTINFO` actualizados. El miembro *flags* contiene información adicional sobre la posición relativa del punto con respecto al control o al ítem.

```
hhi.pt.x=10;
hhi.pt.y=10;
int i;
i = SendDlgItemMessage(hwnd, ID_HEADER1, HDM_HITTEST,
(WPARAM)0, (LPARAM)&hhi);
```

## Mensajes de arrastre de ítems

Los controles de cabecera permiten operaciones de arrastre de columnas. Estas operaciones son automáticas, tan sólo es necesario que el control tenga definido el estilo `HDS_DRAGDROP`.

Cuando el usuario inicie una operación de arrastre, la aplicación recibirá un mensaje de notificación `HDN_BEGINDRAG`. Si la aplicación retorna `FALSE`, el sistema se ocupará de crear la imagen y mostrarla en pantalla a medida que el punto del ratón cambie de posición. Si por el contrario queremos que la aplicación se ocupe de todas las tareas: crear una imagen y actualizar la pantalla cuando el ratón se mueva, tendremos que realizar esas tareas y retornar `TRUE`.

En caso de que queramos evitar el comportamiento automático, mediante el mensaje `HDM_CREATEDRAGIMAGE` podemos obtener una imagen semitransparente, para ello pasaremos el índice del

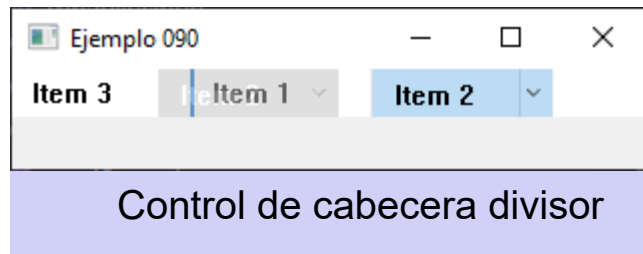


ítem en wParam, y se retornará una lista de imágenes con una única imagen.

Cuando la operación de arrastre concluya, es decir, cuando el usuario libere el botón izquierdo del ratón, se enviará un mensaje de notificación [HDN\\_ENDDRAG](#) a la aplicación. De nuevo, si la aplicación retorna FALSE a este mensaje, el sistema se encargará de situar el ítem en su nueva posición. En caso contrario será la aplicación la encargada de reordenar los ítems, usando el mensaje [HDM\\_SETITEM](#) o el mensaje [HDM\\_SETORDERARRAY](#).

De hecho, si optamos por el funcionamiento automático no será necesario manejar estos mensajes de notificación. El comportamiento por defecto es retornar FALSE en ambos.


En el comportamiento automático se resalta con una línea azul la separación entre ítems donde se insertará el ítem que se está arrastrando. Esto se puede hacer también usando el mensaje [HDM\\_SETHOTDIVIDER](#) cuando sea la aplicación la responsable de procesar los mensajes de draganddrop de ítems.



## Arrastre de divisores

Si el control de cabecera no tiene el estilo HDS\_NOSIZING, el usuario podrá modificar la anchura de un ítem arrastrando el divisor de la derecha de ese ítem.

Windows gestiona automáticamente las operaciones de arrastre de divisores, pero si una aplicación tiene que gestionar estas operaciones tendrá que procesar algunos mensajes de notificación.

La operación de arrastre comienza cuando el usuario hace clic con el botón izquierdo sobre la división entre dos ítems. El programa indica que el cursor está en ese divisor cambiando el cursor  Cursor de divisor. La aplicación recibe un mensaje de

notificación [HDN\\_BEGINTRACK](#). Se siguen enviando mensajes de notificación [HDN\\_TRACK](#) mientras el usuario mantenga pulsado el botón izquierdo y mueva el ratón. Cuando se libere el botón del ratón se envía un mensaje de notificación [HDN\\_ENDTRACK](#).

Además la aplicación recibirá un mensaje de notificación [HDN\\_DIVIDERDBLCLICK](#) cuando el usuario haga doble clic sobre un divisor. De nuevo, no hay un comportamiento automático como respuesta a este mensaje y será la aplicación la responsable de procesarlo. El comportamiento esperado en este caso es ajustar la anchura del ítem a la mínima para mostrar el texto, pero en cada caso esto puede ser diferente.

## Mensajes de filtros

Si el control de cabecera tiene el estilo `HDS_FILTERBAR` se mostrará una segunda línea en cada ítem del control que permite editar textos, números o fechas que se usarán como filtro para la columna.

Cuando se especifica como numérico para el tipo de filtro, sólo se permiten valores enteros, no en coma flotante.

Disponemos de varios mensajes para actualizar filtros, [HDM\\_CLEARFILTER](#) borra el valor del filtro para un ítem determinado. El índice del ítem se especifica en `wParam`. `lParam` no se usa.

El mensaje [HDM\\_EDITFILTER](#) sirve para situar el foco en el cuadro de edición del filtro. En `wParam` se indica el índice del ítem, y en `lParam` se indica qué hacer con el valor del texto si el filtro ya estaba siendo editado por el usuario.

Desde que se modifica el valor de un filtro hasta que se envía a la aplicación un mensaje de notificación [HDN\\_FILTERCHANGE](#) transcurre cierto tiempo. Este tiempo se puede modificar mediante el mensaje [HDM\\_SETFILTERCHANGETIMEOUT](#) indicando en `lParam` el tiempo en milisegundos desde que se modifican los atributos del filtro hasta que se envía la notificación.

Cada vez que el valor de un filtro se modifica se envía un mensaje de notificación `HDN_FILTERCHANGE` a la aplicación. Esto se hace incluso durante la edición del valor del filtro, siempre que no haya transcurrido el tiempo de espera entre modificaciones sucesivas.

Por ejemplo, el usuario está editando un filtro, y el tiempo de espera (timeout) está establecido en un segundo (1000 ms), mientras el usuario está escribiendo, si deja de hacerlo más de un segundo, se enviará un mensaje de notificación `HDN_FILTERCHANGE`, si no llega a pasar ese tiempo entre actualizaciones del valor, no se enviarán mensajes de notificación.

Hay que tener en cuenta que aplicar un filtro puede ser una tarea que requiera mucho tiempo de procesador y de actualización de pantalla, por lo que especificar un tiempo de espera permite que el usuario pueda modificar el valor del filtro sin que la aplicación reciba la notificación de que el valor del filtro ha sido modificado. Estos mensajes de notificación se envían durante la edición del filtro, y no sólo cuando el usuario de por terminada la edición pulsado return, o haciendo click en otra zona de la pantalla.

También se envían a la aplicación mensajes de notificación cuando se inicia la edición de un filtro, `HDN_BEGINFILTEREDIT`, y cuando termina `HDN_ENDFILTEREDIT`.

Por último, el mensaje de notificación `HDN_FILTERBTNCLICK` se envía a la aplicación cuando se pulsa sobre el icono del filtro de un ítem (el icono del embudo). También se envía esta notificación como respuesta a un mensaje `HDM_SETITEM`.

De nuevo, el sistema no tiene un comportamiento definido para este botón, y será la aplicación la encargada de responder adecuadamente a éste mensaje, si lo considera necesario.

## Indicativos de orden

Cuando un control de cabecera esté asociado a un listview, cada columna contendrá una lista de valores. Ya hemos visto que

podemos filtrar esos valores, y además podremos necesitar ordenarlos. Cuando los valores de una columna estén ordenados pueden aparecer en orden ascendente o descendente. Cada item permite mostrar una marca para indicar el orden. Estas marcas se activan mediante el miembro *fmt* de la estructura [HDITEM](#), añadiendo los valores `HDF_SORTUP` o `HDF_SORTDOWN`, respectivamente.

Podemos hacer esto al insertar cada item, o bien modificándolo mediante un mensaje [HDM\\_SETITEM](#).

```
HDITEM hdi;
int iItem = 1;
...
hdi.mask = HDI_FORMAT;
SendDlgItemMessage(hwnd, ID_HEADER1, HDM_GETITEM,
(WPARAM)iItem, (LPARAM)&hdi);
hdi.fmt |= HDF_SORTUP;
SendDlgItemMessage(hwnd, ID_HEADER1, HDM_SETITEM,
(WPARAM)iItem, (LPARAM)&hdi);
```

## Mensajes de foco de teclado

Una de las columnas del control de cabecera tendrá el foco del teclado, que se visualizará resaltando el fondo del item. Podemos recuperar el número del item que tiene el foco actualmente usando el mensaje [HDM\\_GETFOCUSEDITEM](#). Este mensaje no requiere parámetros. También podemos asignar el foco al item que queramos mediante el mensaje [HDM\\_SETFOCUSEDITEM](#) indicando el item en el parámetro `lParam`.

## Mensajes de situación en ventana

El mensaje [HDM\\_LAYOUT](#) calcula las dimensiones y posición correctas para un control de cabecera a partir de un rectángulo

determinado. El parámetro `wParam` no se usa y en `lParam` se pasa un puntero a una estructura [HDLAYOUT](#). Al enviar este mensaje, el miembro `prc` debe contener las coordenadas del rectángulo, y devolverá en el miembro `pwpos` el tamaño y posición calculados para el control.

El mensaje [HDM\\_GETITEMRECT](#) sirve para recuperar el rectángulo que ocupa un determinado ítem del control. En `wParam` se debe indicar el índice del ítem y en `lParam` un puntero a una estructura [RECT](#) que recibirá la información del rectángulo.

## Botón de desplegar

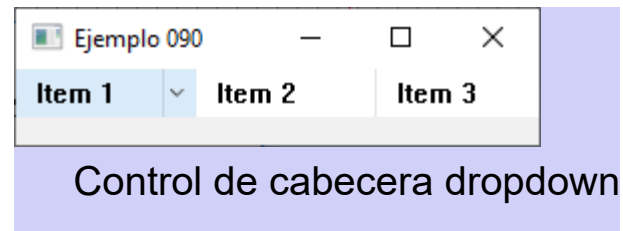
Cuando se crea un control de cabecera con el estilo `HDS_BUTTONS` y el ítem incluya la bandera de formato `HDF_SPLITBUTTON`, la

aplicación recibirá un mensaje de notificación [HDN\\_DROPDOWN](#) cuando el usuario haga clic con el ratón en el icono de despliegue. El icono permanecerá oculto, y sólo se mostrará cuando el cursor del ratón esté sobre el ítem.

No hay comportamiento por defecto definido para este mensaje, será el programador el responsable de mostrar en pantalla el resultado de pulsar ese botón. Por ejemplo, las hojas de cálculo despliegan una ventana para definir filtros, pero cada aplicación puede requerir un comportamiento diferente.

Existen dos mensajes que pueden resultar útiles para procesar éste mensaje:

- [HDM\\_GETITEMDROPDOWNRECT](#) obtiene un rectángulo en el que se muestra el icono de despliegue. En `wParam` se indica el índice del ítem, y en `lParam` se pasa un puntero a una estructura [RECT](#) que recibirá las coordenadas del rectángulo.
- [HDM\\_GETITEMRECT](#) obtiene las coordenadas del rectángulo que contiene el ítem. En `wParam` se indica el índice del ítem, y



en `LPARAM` se pasa un puntero a una estructura `RECT` que recibirá las coordenadas del rectángulo.


## Cajas de chequeo

Para cada ítem se puede añadir una caja de chequeo. Para ello el control debe tener el estilo `HDS_CHECKBOXES`, y además, el ítem debe tener el flag `HDF_CHECKBOX`. Adicionalmente se puede añadir el flag `HDF_CHECKED` para indicar que la caja esté marcada.

Se enviará un mensaje de notificación `HDN_ITEMSTATEICONCLICK` a la aplicación cuando el usuario pulse sobre la caja de chequeo. Para que este mensaje se envíe, el control también debe tener el estilo `HDS_BUTTONS`.

```
case HDN_ITEMSTATEICONCLICK:
    iItem = pnmhdr->iItem;
    hdi.mask = HDI_FORMAT;
    SendDlgItemMessage(hwnd, ID_HEADER1,
HDM_GETITEM, (LPARAM)iItem, (LPARAM)&hdi);
    if(hdi.fmt & HDF_CHECKED) hdi.fmt =
(UINT)hdi.fmt & ~HDF_CHECKED;
    else hdi.fmt |= HDF_CHECKED;
    hdi.mask = HDI_FORMAT;
    SendDlgItemMessage(hwnd, ID_HEADER1,
HDM_SETITEM, (LPARAM)iItem, (LPARAM)&hdi);
    break;
```

## Overflow

Si se crea el control de cabecera con el estilo `HDS_OVERFLOW`, si todos los ítems no pueden ser visualizados en pantalla porque el ancho de la ventana es insuficiente, se mostrará un botón  Botón de overflow a la derecha del control.

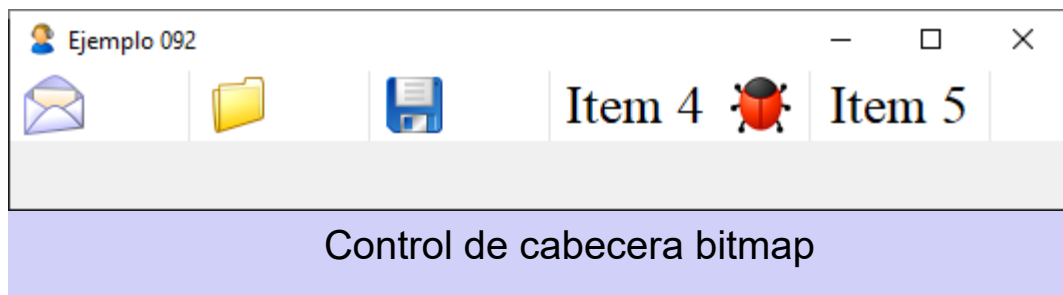
Podemos recuperar el rectángulo delimitador de ese botón mediante el mensaje `HDM_GETOVERFLOWRECT`, pasando en

lParam un puntero a una estructura [RECT](#) que recibirá la información del rectángulo.

Si el usuario pulsa el botón de overflow se enviará un mensaje de notificación [HDN\\_OVERFLOWCLICK](#). Tampoco existe un comportamiento predefinido para este mensaje, de modo que de nuevo será la aplicación la encargada de procesarlo.

## Mensajes de gestión de mapas de bits

Es  
posible  
añadir  
imágenes  
a



los items del control de cabecera, ya sea en solitario o combinados con texto. Para insertar un mapa de bits hay que indicar en el miembro *mask* de [HDITEM](#) el valor [HDF\\_BITMAP](#) o [HDF\\_BITMAP\\_ON\\_RIGHT](#) (si queremos que el mapa de bits aparezca a la derecha del texto), y asignar un mapa de bits al miembro *hbm*. En este ejemplo cargaremos el mapa de bits desde un recurso:

```
hdi.fmt = HDI_FORMAT | HDI_WIDTH | HDF_BITMAP;  
hdi.hbm = LoadBitmap(hInstance, bm);  
...  
SendMessage(hwndHeader, HDM_INSERTITEM, (WPARAM)  
iDespuesDe, (LPARAM) &hdi);
```

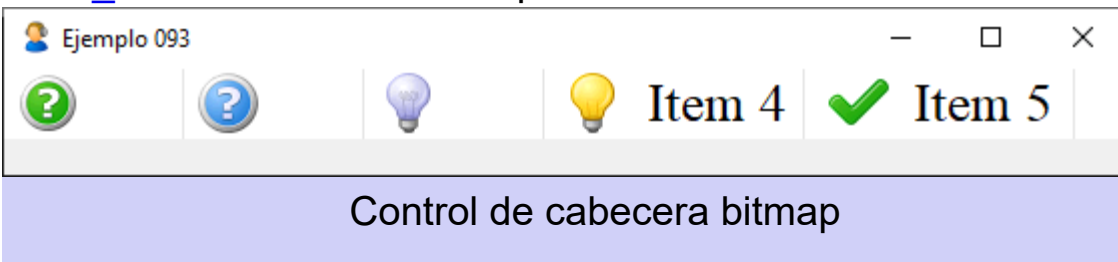
Si además queremos añadir un texto, habrá que añadir al miembro *mask* el valor [HDI\\_TEXT](#) e indicar el formato preferido en el miembro *fmt*:

```

        hdi.mask |= HDI_TEXT;
        hdi.fmt = HDI_FORMAT | HDI_WIDTH | HDI_TEXT;
        hdi.fmt = HDI_FORMAT | HDI_WIDTH |
HDF_BITMAP_ON_RIGHT | HDF_STRING | HDF_CENTER;
        hdi.pszText = texto;
        hdi.hbm = LoadBitmap(hInstance, bm);
        ...
        SendMessage(hwndHeader, HDM_INSERTITEM, (WPARAM)
iDespuesDe, (LPARAM) &hdi);

```

Con el mensaje [HDM\\_SETBITMAPMARGIN](#) obtendremos el valor del margen alrededor del mapa de bits, y con [HDM\\_GETBITMAPMARGIN](#) podemos modificar ese valor.



También podemos usar imágenes de una lista de imágenes. Para ello asignaremos una lista de imágenes al control mediante el mensaje [HDM\\_SETIMAGELIST](#). En wParam se indicará el tipo de lista de imágenes, normal o de estado. En lParam pasaremos el manipulador de la lista de imágenes.

Para recuperar la lista de imágenes actualmente asignada al control, si la hay, se usa el mensaje [HDM\\_GETIMAGELIST](#) en wParam indicaremos que tipo de lista queremos recuperar, normal o de estado. lParam no se usa, y se retorna un manipulador de la lista de imágenes.

Para usar imágenes de una lista primero hay que asignar una lista de imágenes al control. Después, para cada columna, se indicará en *mask* el valor `HDI_IMAGE`, en *fmt* el valor `HDF_IMAGE`, y en el miembro *ilimage* el índice de la imagen a mostrar:

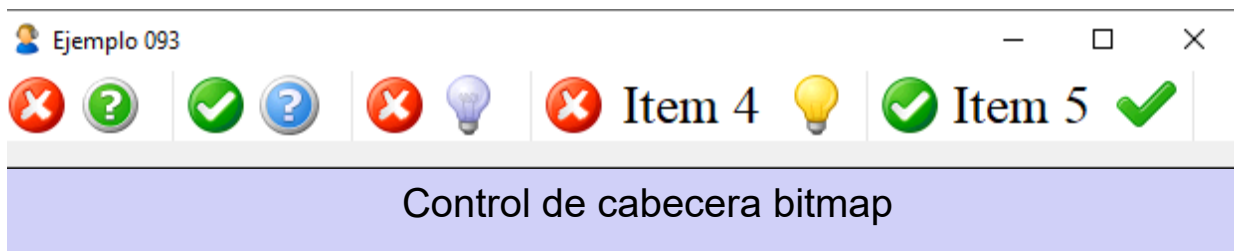
```
// Asignar lista de imágenes:
```



```

        SendDlgItemMessage(hwnd, ID_HEADER1,
        HDM_SETIMAGELIST, (WPARAM)HDSIL_NORMAL, (LPARAM) hIm1);
        SendDlgItemMessage(hwnd, ID_HEADER1,
        HDM_SETIMAGELIST, (WPARAM)HDSIL_STATE, (LPARAM) hIm1Estado);
        ...
        hdi.mask = HDI_FORMAT | HDI_WIDTH | HDI_IMAGE;
        hdi.fmt = HDF_IMAGE;
        hdi.iImage = i1; // índice de imagen dentro de la
        lista
        ...
        SendMessage(hwndHeader, HDM_INSERTITEM, (WPARAM)
        iDespuesDe, (LPARAM) &hdi);

```



La lista de imágenes de estado se usa como sustitución de las cajas de chequeo. La primera imagen de la lista se usará cuando la caja de chequeo no esté marcada, y la segunda para cuando lo esté. Por supuesto, el control debe tener el estilo `HDS_CHECKBOXES`, y el ítem la bandera `HDF_CHECKBOX`. También se debe procesar el mensaje de notificación `HDN_ITEMSTATEICONCLICK`.

La bandera `HDF_BITMAP_ON_RIGHT` también se puede aplicar a imágenes procedentes de una lista de imágenes.

## Mensajes de codificación de caracteres

Para el soporte de Unicode disponemos de dos mensajes `HDM_GETUNICODEFORMAT`, sin parámetros. Un valor de retorno cero indica que el control está usando caracteres ANSI, en caso contrario estará usando caracteres Unicode.

Para cambiar el conjunto de caracteres a usar en el control se puede usar el mensaje `HDM_SETUNICODEFORMAT`, indicando en

wParam un valor nulo para usar caracteres ANSI o un valor distinto de cero para usar Unicode.

## Acción del ratón sobre ítems

Se enviará un mensaje de notificación [HDN\\_ITEMCLICK](#) cuando el usuario haga clic con el ratón sobre un ítem, y un mensaje [HDN\\_ITEMDBLCLICK](#) si hace doble clic.

También se recibirá un mensaje de notificación [NM\\_RCLICK](#) cuando el usuario haga clic con el botón derecho sobre uno de los ítems del control. Este mensaje no es específico del control de cabecera, y se envía para otros controles también, por lo que la aplicación deberá determinar de qué tipo de control proviene el mensaje.

No hay comportamiento por defecto para estos mensajes, de modo que debe ser la aplicación la que responda a ellos de la forma que se considere oportuna.

## Notificaciones de modificación de ítem

Cada vez que se vayan a modificar las propiedades de un ítem se envían dos mensajes de notificación a la aplicación. El primero es un mensaje [HDN\\_ITEMCHANGING](#) que se envía justo antes de que se modifique el ítem. Esto da una oportunidad a la aplicación para permitir o no los cambios, o ajustar los nuevos valores a los permitidos por el diseño del programa.

El mensaje de notificación [HDN\\_ITEMCHANGED](#) se envía a la aplicación después de que los cambios se hayan completado. Esto permite que la aplicación pueda procesar el resultado de aplicar esos cambios, por ejemplo, en un ListView actualizaría los datos de la lista.

## Pulsaciones de tecla

Se envían mensaje de notificación [HDN\\_ITEMKEYDOWN](#) cuando el usuario pulsa alguna tecla estando activo el control. No se envían notificaciones con todas las teclas.

## Inserción con datos incompletos

Al insertar un ítem en un control de cabecera podemos omitir el texto y el índice de la imagen de la lista de imágenes. En el primer caso usaremos el valor `LPSTR_TEXTCALLBACK` en lugar del texto para asignar el miembro *pszText* de la estructura [HDITEM](#). Para el índice de la imagen usaremos el valor `I_IMAGECALLBACK` para el miembro *iImage*.

Cuando el control necesite la información ausente la pedirá a la aplicación mediante un mensaje de notificación [HDN\\_GETDISPINFO](#). La aplicación debe consultar el valor del miembro *mask* de la estructura [HDITEM](#) que recibirá como un puntero en *lParam* para determinar qué valores está pidiendo el control, y asignará los miembros correspondientes. Si además añade el valor `HDI_DI_SETITEM` al *mask*, el control almacenará esos valores y no volverá a solicitarlos.

```
NMHDDISPINFO* pdi;
char* szTextoItem = "Prueba";
...
case WM_NOTIFY:
    pnmhdr = (LPNMHEADER)lParam;
    switch(pnmhdr->hdr.code) {
        case HDN_GETDISPINFO:
            pdi = (NMHDDISPINFO*)lParam;
            if(pdi->mask & HDI_IMAGE) {
                pdi->iImage = 4;
            }
            if(pdi->mask & HDI_TEXT) {
                strcpy(pdi->pszText, szTextoItem);
            }
            pdi->mask |= HDI_DI_SETITEM;
            return 0;
    }
```

```
...  
}
```

## Otros mensajes de notificación

Otros mensajes de notificación que se pueden enviar a la aplicación, pero que no son exclusivos de este tipo de controles son [NM\\_CUSTOMDRAW](#) que sirve para personalizar la apariencia del control.

### Ejemplo 91

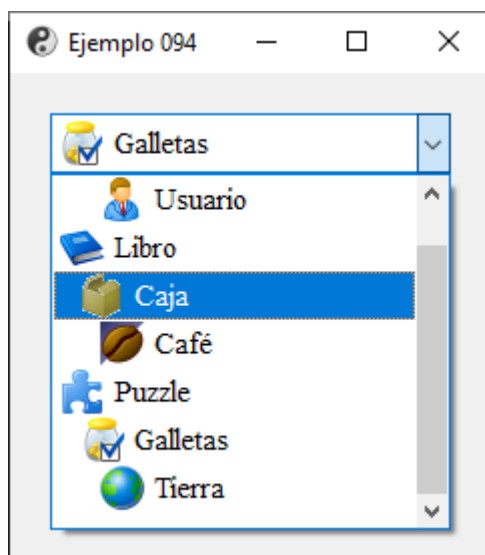
### Ejemplo 92

### Ejemplo 93

# Capítulo 54 Control ComboBoxEx

Un control ComboBoxEx es, como su nombre indica, un control ComboBox extendido. La mayor diferencia con los ComboBox que hemos visto hasta ahora es que permite mostrar imágenes para cada item.

Por lo demás, siguen siendo un controles ComboBox, es decir, contienen una caja de edición y una lista desplegable con las posibles opciones. Mantiene los estilos de control de los ComboBoxes básicos:



Ejemplo de control ComboBoxEx

- CBS\_DROPDOWN: el control de edición asociado permite introducir valores que no estén en la lista y una lista desplegable.
- CBS\_DROPDOWNLIST: el control de edición está deshabilitado y sólo se pueden seleccionar valores que ya estén en la lista desplegable.
- CBS\_SIMPLE: el cuadro de edición se muestra separado de la lista, y la lista siempre se muestra desplegada. Este estilo no funciona bien con algunas características de los ComboBoxEx.

Como en todos los controles comunes que estamos viendo, hay que asegurarse de que la DLL ha sido cargada invocando a la función [InitCommonControlsEx](#) indicando el valor de bandera

ICC\_USEREX\_CLASSES en el miembro *dwICC* de la estructura [INITCOMMONCONTROLSEX](#) que pasaremos como parámetro.

```
INITCOMMONCONTROLSEX icCE;  
...  
icCE.dwSize = sizeof(INITCOMMONCONTROLSEX);  
icCE.dwICC = ICC_USEREX_CLASSES;  
InitCommonControlsEx(&icCE);
```

## Insertar durante la ejecución

Del mismo modo que hemos visto para otros controles, también es posible insertar controles ComboBoxEx durante la ejecución. Tan sólo hay que crear una ventana de la clase "WC\_COMBOBOXEX". Para hacerlo usaremos las funciones [CreateWindow](#) o [CreateWindowEx](#).

```
case WM_CREATE:  
    hInstance = ((LPCREATESTRUCT)lParam)->hInstance;  
    hCtrl = CreateWindowEx(0, WC_COMBOBOXEX, NULL,  
        WS_BORDER | WS_VISIBLE | WS_CHILD |  
CBS_DROPDOWN,  
        0, 0, 200, 200,  
        hwnd, (HMENU)ID_COMBOBOXEX,  
        hInstance, NULL);  
    break;
```

En el parámetro *hMenu* indicaremos el identificador del control y elegiremos los estilos y dimensiones adecuados para nuestro caso.

Hay que recordar que cuando se insertan controles durante la ejecución, la fuente por defecto es system. Si queremos cambiarla habrá que crear una fuente y asignársela al control usando un mensaje [WM\\_SETFONT](#). Hay que recordar liberar recursos antes de cerrar la aplicación, usando [DeleteObject](#).

```

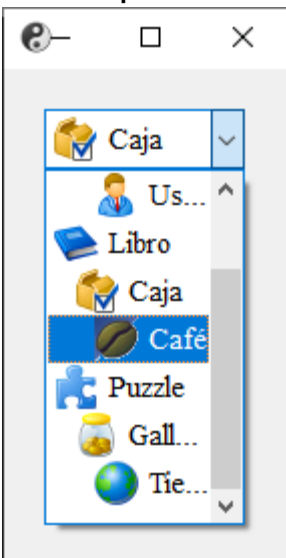
static HFONT hFont;
...
case WM_CREATE:
    hFont = CreateFont(18, 0, 0, 0, 300, FALSE,
FALSE, FALSE,
                        DEFAULT_CHARSET, OUT_TT_PRECIS,
CLIP_DEFAULT_PRECIS,
                        PROOF_QUALITY, DEFAULT_PITCH | FF_ROMAN,
"Times New Roman");
    // Asignamos la fuente a nuestro gusto.
    SendMessage(hCtrl, WM_SETFONT, (WPARAM)hFont,
MAKELPARAM(TRUE, 0));
...
case WM_DESTROY:
    DeleteObject(hFont);
    break;

```

## Estilos

Siguen disponibles los estilos de los controles ComboBox, ver [capítulo 43](#) para más detalles.

Además, existen otros [estilos extendidos](#), que básicamente sirven para limitar algunas de las nuevas características.



ComboBoxEx Elipsis

- CBES\_EX\_CASESENSITIVE: las búsquedas de cadenas en la lista distinguen mayúsculas de minúsculas.
- CBES\_EX\_NOEDITIMAGE: no se muestra imagen de ítem en la caja de edición ni en la lista.
- CBES\_EX\_NOEDITIMAGEINDENT: equivale a CBES\_EX\_NOEDITIMAGE
- CBES\_EX\_NOSIZELIMIT: Permite que el tamaño vertical del control ComboBoxEx sea más pequeño que el del control ComboBox incluido.
- CBES\_EX\_PATHWORDBREAKPROC: se usarán los caracteres '/', '\', y '.' como

delimitadores de palabra. Esto ayuda a moverse por palabras en nombres de fichero y URLs

- **CBES\_EX\_TEXTENDELLIPSIS**: cuando el texto de un ítem no quepa en el ancho del control, en lugar de cortarse se sustituye el final por puntos suspensivos.

Para asignar y retirar estilos extendidos a un control ComboBoxEx se usa el mensaje **CBEM\_SETTEXTENDEDSTYLE** en wParam se indica una máscara de qué estilos queremos asignar o retirar y en lParam qué estilos queremos modificar. Por ejemplo, si queremos asignar **CBES\_EX\_CASESENSITIVE** y quitar **CBES\_EX\_TEXTENDELLIPSIS**, usaremos **CBES\_EX\_CASESENSITIVE** en wParam y **CBES\_EX\_CASESENSITIVE | CBES\_EX\_TEXTENDELLIPSIS** en lParam. Esto hace que se puedan asignar o retirar estilos extendidos sin necesidad de leer los estilos extendidos actuales.

```
SendMessage(hCtrl, CBEM_SETTEXTENDEDSTYLE,  
(WPARAM)CBES_EX_CASESENSITIVE,  
          (LPARAM)CBES_EX_CASESENSITIVE |  
          CBES_EX_TEXTENDELLIPSIS);
```

Para recuperar los estilos actuales de un control ComboBoxEx se usa el mensaje **CBEM\_GETTEXTENDEDSTYLE**.

## Lista de imágenes

Cada ítem en un control ComboBoxEx tiene asociadas tres imágenes, una para mostrar normalmente, otra para mostrar cuando el ítem está activo y una tercera que se usa para superponer. El mensaje **CBEM\_SETIMAGELIST** sirve para asignar una lista de imágenes a un control ComboBoxEx.

Para recuperar la lista de imágenes asociada a un control ComboBoxEx se usa el mensaje **CBEM\_GETIMAGELIST**, sin



parámetros.

Es importante eliminar la lista de imagenes antes de terminar la aplicación.

```
HIMAGELIST hIml;
HWND hCtrl;
HBITMAP hbmp;

...
    case WM_CREATE:
        hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
        hIml = ImageList_Create(24, 24,
ILC_COLOR16|ILC_MASK, 19, 4);
        hbmp = LoadBitmap(hInstance, "imagenes");
        ImageList_AddMasked(hIml, hbmp,
RGB(255,255,255));
        DeleteObject(hbmp);
        hCtrl = CreateWindowEx(0, WC_COMBOBOXEX, NULL,
            WS_BORDER | WS_VISIBLE | WS_CHILD |
CBS_DROPDOWN | CBS_SORT,
            0, 0, 0, 200,
            hwnd, (HMENU)ID_COMBOBOXEX,
            hInstance, NULL);
        SendMessage(hCtrl, CBEM_SETIMAGELIST, 0,
(LPARAM)hIml);
    ...
    case WM_DESTROY:
        hIml = (HIMAGELIST)SendMessage(hCtrl,
CBEM_GETIMAGELIST, 0, 0);
        ImageList_Destroy(hIml);
    ...
```

## Insertar items

Para insertar items en un control ComboBoxEx se usa el mensaje **CBEM\_INSERTITEM**. En lParam hay que pasar un puntero a una estructura **COMBOBOXEXITEM** con la información correspondiente al item.

En el miembro *mask* de la estructura se deben activar las banderas que indican qué miembros de la estructura contienen

valores válidos. En *iItem* el índice del item.

En el miembro *iItem* se especifica la posición de inserción. El valor debe estar entre 0 y el número de items en el control. Si se especifica un valor mayor, la operación de inserción fallará.

Para insertar el item en la última posición se puede usar el valor retornado por el mensaje [CB\\_GETCOUNT](#).

Para asignar un texto al item hay que asignar valores a los miembros *pszText*. El miembro *cchTextMax* se ignora cuando se está insertando un item.

El miembro *iImage* es el índice de la imagen dentro de la lista de imágenes asignada al control que se muestra cuando el item no esté seleccionado.

El miembro *iSelectedImage* es el índice de la imagen dentro de la lista de imágenes asignada al control que se muestra cuando el item esté seleccionado.

El miembro *iOverlay* se supone que es para generar una imagen superponiéndola a otra existente (presumiblemente *iImage*), sin embargo no parece que funcione y la documentación al respecto es muy incompleta.

El miembro *iIndent* sirve para añadir espacios a la izquierda del item. Cada espacio equivale a 10 pixels.

Finalmente, el miembro *IPParam* permite almacenar un valor que puede usar la aplicación para su funcionamiento.

```
BOOL InsertarItem(HWND hCtrl, int iItem, int imagen, int
imagensel, int indent, char *texto) {
    COMBOBOXEXITEM cbei;

    cbei.mask = CBEIF_TEXT | CBEIF_INDENT | CBEIF_IMAGE |
CBEIF_SELECTEDIMAGE;
    cbei.iItem          = iItem;
    cbei.pszText         = texto;
    cbei.iImage          = imagen;
    cbei.iSelectedImage = imagensel;
    cbei.iIndent         = indent;

    // Intenta insertar el item, retorna FALSE si falla.
```

```

        if (SendMessage(hCtrl, CBEM_INSERTITEM, 0, (LPARAM) &cbei)
== -1)
            return FALSE;
        return TRUE;
    }

```

Cuando se insertan items, los valores de algunos miembros de la estructura **COMBOBOXEXITEM** pueden omitirse y se podrán asignar cuando el control necesite acceder a ellos.

En el caso de las imágenes, *image*, *iSelectedImage* e *iOverlay* se puede asignar el valor **I\_IMAGECALLBACK** en lugar de un índice o un valor concreto. En el caso de *pszText* se puede usar el valor **LPSTR\_TEXTCALLBACK**.

Cuando el control necesite mostrar uno de esos items requerirá la información a la aplicación mediante un mensaje de notificación **CBEN\_GETDISPINFO**.

Este mensaje de notificación recibirá a través de *lParam* un puntero a una estructura **NMCOMBOBOXEX**, que a su vez contiene una estructura **COMBOBOXEXITEM**. En esa estructura, el miembro *mask* nos indicará que miembros de la estructura debemos asignar antes de retornar con un valor nulo.

Si además añadimos el valor **CBEIF\_DI\_SETITEM** a *mask*, el control almacenará la información suministrada, y no volverá a solicitarla.

En este ejemplo se calculan los índices de la imágenes y los márgenes de cada item en función del valor de *iIndex*:

```

BOOL InsertarItem(HWND hCtrl, int iItem) {
    COMBOBOXEXITEM cbei;

    cbei.mask = CBEIF_TEXT | CBEIF_INDENT | CBEIF_IMAGE |
CBEIF_SELECTEDIMAGE;
    cbei.iItem          = iItem;
    cbei.pszText        = LPSTR_TEXTCALLBACK;
    cbei.iImage         = I_IMAGECALLBACK;
    cbei.iSelectedImage = I_IMAGECALLBACK;
}

```

```

        // Intenta insertar el item, retorna FALSE si falla.
        if(SendMessage(hCtrl, CBEM_INSERTITEM, 0, (LPARAM)&cbei)
== -1)
            return FALSE;
        return TRUE;
    }
    ...
    NMCOMBOBOXEX* nmCBE;
    ...
    case WM_NOTIFY:
        pnmhdr = (LPNMHEADER)lParam;
        switch(pnmhdr->hdr.code) {
            case CBEN_GETDISPINFO:
                if( nmCBE->ceItem.mask & CBEIF_IMAGE)
                    nmCBE->ceItem.iImage = 1; // Imagen
1 de lista
                    if( nmCBE->ceItem.mask &
CBEIF_SELECTEDIMAGE)
                        nmCBE->ceItem.iSelectedImage = 2; //
Imagen 2 de lista
                    if( nmCBE->ceItem.mask & CBEIF_TEXT)
                        strcpy(nmCBE->ceItem.pszText,
"Desconocido");
                    nmCBE->ceItem.mask |= CBEIF_DI_SETITEM;
                    return 0;
                    break;
        }

```

También se envía a la aplicación un mensaje de notificación **CBEN\_INSERTITEM** cada vez que se inserte un ítem en el control. En este caso también se recibirá a través de *LPARAM* un puntero a una estructura **NMCOMBOBOXEX** con los datos del ítem insertado.

## Modificar un ítem

Es posible modificar los atributos de un ítem que ya esté en la lista, para ello usaremos el mensaje **CBEM\_SETITEM**. El comportamiento es similar al de insertar un ítem, pasando en *LPARAM* un puntero a una estructura **COMBOBOXEXITEM** con los miembros asignados con los nuevos valores del ítem y el índice del ítem en el miembro *item*.

```
COMBOBOXEXITEM cbeitem;
...
    cbeitem.mask = CBEIF_TEXT;
    cbeitem.iItem = 4;
    cbeitem.pszText = "Modificado";
    SendMessage(hCtrl, CBEM_SETITEM, 0,
(LPARAM) &cbeitem);
```

## Obtener información de un ítem

Para recuperar información sobre un ítem usaremos el mensaje **CBEM\_GETITEM** pasando en *lParam* un puntero a una estructura **COMBOBOXEXITEM**, en la que iniciaremos los miembros *iItem* para indicar de qué ítem queremos recuperar datos, y *mask* para determinar que miembros de la estructura queremos obtener.

## Eliminar un ítem

Para eliminar ítems se usa el mensaje **CBEM\_DELETEITEM**. En *wParam* se determina el número del *iItem* a borrar.

```
SendMessage(hCtrl, CBEM_DELETEITEM, (WPARAM)1, 0);
```

Cuando se borra un ítem se envía un mensaje de notificación **CBEN\_DELETEITEM** a la aplicación. En *lParam* se envía un puntero a una estructura **NMCOMBOBOXEX** con la información sobre el ítem eliminado.

## Edición de valores

Si el control no tiene el estilo **CBS\_DROPDOWNLIST**, la caja de edición estará activa.

Cada vez que el usuario pulsa sobre la caja de edición, o que se active por otra causa (TAB o atajo de teclado o que se haya pulsado el botón de despliegue de la lista), la aplicación recibirá un mensaje de notificación `CBEN_BEGINEDIT`. A partir de ese momento estaremos en proceso de edición. Cuando la edición termine se enviará a la aplicación recibirá un mensaje de notificación `CBEN_ENDEDIT`.

Con el mensaje de notificación `CBEN_ENDEDIT`, en `lParam` se envía un puntero a una estructura `NMCBEENDEDIT`. En esta estructura, el miembro *fChanged* nos indicará si el valor actual de la caja de edición ha sido modificado. *iNewSelection* contendrá el índice del ítem de la lista, siempre que el contenido actual de la caja de edición esté en la lista. En caso contrario será -1. *szText* contiene el valor de la caja de edición y *iWhy* nos indica el motivo por el que se ha dado por concluida la edición, ya sea pérdida de foco, despliegue de la lista o pulsación de `ESC` o `INTRO`.

```
NMCBEENDEDIT* nmCEE;
...
case WM_NOTIFY:
    pnmhdr = (LPNMHEADER)lParam;
    switch(pnmhdr->hdr.code) {
        case CBEN_BEGINEDIT:
            return 0;
        case CBEN_ENDEDIT:
            nmCEE = (NMCBEENDEDIT*)lParam;
            if(nmCEE->fChanged && nmCEE->
iNewSelection == -1) {
                InsertarItem(hCtrl, -1, 17, 8, 2,
nmCEE->szText);
                c = SendMessage(hCtrl, CB_GETCOUNT,
0, 0);
                SendMessage(hCtrl, CB_SETCURSEL,
(WPARAM)c, 0);
            }
            return 0;
        }
    }
    break;
```

También disponemos de un mensaje para averiguar si el contenido de la caja de edición [CBEM\\_HASEDITCHANGED](#), pero sólo funciona si se usa antes de que se envíe el mensaje de notificación [NMCBEENDEEDIT](#), ya que si se envía después siempre devuelve FALSE.

## Ejemplo 94

### Ficheros de recursos

No existe un tipo de control específico para crear ComboBoxEx en un fichero de recursos, de modo que siempre deberemos insertar estos controles en ejecución. Podemos, sin embargo, usar un control ComboBox para situar el control si usamos un editor de recursos como ResEdit, y convertir las coordenadas de diálogo a coordenadas de pantalla usando la función [MapDialogRect](#):

```
//
// Dialog resources
//
LANGUAGE LANG_NEUTRAL, SUBLANG_NEUTRAL
DialogEx DIALOG 0, 0, 321, 57
STYLE DS_3DLOOK | DS_CENTER | DS_MODALFRAME | DS_SHELLFONT |
WS_CAPTION | WS_VISIBLE | WS_POPUP | WS_SYSMENU
CAPTION "ComboBoxEx"
FONT 8, "Helv"
{
/* COMBOBOX usado como plantilla para COMBOBOXEX, comentado
posteriormente para ser insertado en ejecución
COMBOBOX ID_COMBOBOXEX, 8, 9, 221, 81,
CBS_DROPDOWN | CBS_HASSTRINGS | CBES_EX_CASESENSITIVE,
WS_EX_LEFT */
PUSHBUTTON "Cancel", IDCANCEL, 259, 27, 50, 14, 0,
WS_EX_LEFT
DEFPUSHBUTTON "OK", IDOK, 259, 10, 50, 14, 0,
WS_EX_LEFT
}
```

## Ejemplo de procedimiento de diálogo:

```
BOOL CALLBACK DialogProcedure(HWND hwndDlg, UINT uMsg,
WPARAM wParam, LPARAM lParam) {
    HIMAGELIST hIml;
    HBITMAP hbmp;
    HFONT hfont;
    RECT re;

    switch(uMsg) {
        case WM_INITDIALOG:
            hIml = ImageList_Create(24, 24,
ILC_COLOR16|ILC_MASK, 19, 4);
            hbmp = LoadBitmap((HINSTANCE)lParam,
"imagenes");
            ImageList_AddMasked(hIml, hbmp,
RGB(255,255,255));
            DeleteObject(hbmp);
            hfont = CreateFont(-11, 0, 0, 0, 0, FALSE,
FALSE, FALSE, 1, 0, 0, 0, 0, ("Ms Shell Dlg"));

            re.top = 8;
            re.left = 9;
            re.bottom = 221;
            re.right = 81;
            MapDialogRect(hwndDlg, &re);

            CreateWindowEx(0, WC_COMBOBOXEX, NULL,
                WS_BORDER | WS_VISIBLE | WS_CHILD |
WS_TABSTOP | CBS_DROPDOWN | CBS_SORT | ES_WANTRETURN,
                re.top, re.left, re.bottom, re.right,
                hwndDlg, (HMENU)ID_COMBOBOXEX,
                (HINSTANCE)lParam, NULL);
            // Asignamos la fuente a nuestro gusto.
            SendDlgItemMessage(hwndDlg, ID_COMBOBOXEX,
WM_SETFONT, (WPARAM)hfont, MAKELPARAM(FALSE, 0));
            SendDlgItemMessage(hwndDlg, ID_COMBOBOXEX,
CBEM_SETIMAGELIST, 0, (LPARAM)hIml);

            // Insertar items:
            InsertarItem(hwndDlg, ID_COMBOBOXEX, 0, 9, 0, 0,
"Reloj");
            InsertarItem(hwndDlg, ID_COMBOBOXEX, 1, 10, 1,
1, "Bateria");
            InsertarItem(hwndDlg, ID_COMBOBOXEX, 2, 11, 2,
```



```

2, "Usuario");
    InsertarItem(hwndDlg, ID_COMBOBOXEX, 3, 12, 3,
0, "Libro");
    InsertarItem(hwndDlg, ID_COMBOBOXEX, 4, 13, 4,
1, "Caja");
    InsertarItem(hwndDlg, ID_COMBOBOXEX, 5, 14, 5,
2, "Café");
    InsertarItem(hwndDlg, ID_COMBOBOXEX, 6, 15, 6,
0, "Puzzle");
    InsertarItem(hwndDlg, ID_COMBOBOXEX, 7, 16, 7,
1, "Galletas");
    InsertarItem(hwndDlg, ID_COMBOBOXEX, 8, 17, 8,
2, "Tierra");

    SendDlgItemMessage(hwndDlg, ID_COMBOBOXEX,
CB_SETCURSEL, (WPARAM)0, 0);
    return TRUE;

...
}
}

```

## Mensajes de formato de caracteres

Se puede cambiar la bandera de formato Unicode usando el mensaje [CBEM\\_SETUNICODEFORMAT](#) indicando en wParam el valor de la bandera, cero para caracteres ANSI y distinto de cero para caracteres Unicode.

El mensaje [CBEM\\_GETUNICODEFORMAT](#) obtiene el valor de la bandera de formato Unicode.

## Controles base

Los controles ComboBoxEx constan de dos controles base: un control de edición y un ComboBox. Es posible obtener un manipulador de cada uno de ellos. Para el control ComboBox se usa el mensaje [CBEM\\_GETCOMBOCONTROL](#) y para el control de edición un mensaje [CBEM\\_GETEDITCONTROL](#),

## Operaciones de arrastre

Si la aplicación implementa operaciones de drag and drop al recibir un mensaje de notificación [CBEN\\_DRAGBEGIN](#) debe iniciar una de ellas.

Ya hablamos algo sobre este tipo de operaciones en el [capítulo 48](#), pero probablemente volvamos a dedicar tiempo a este tipo de funciones.

## Temas de Windows

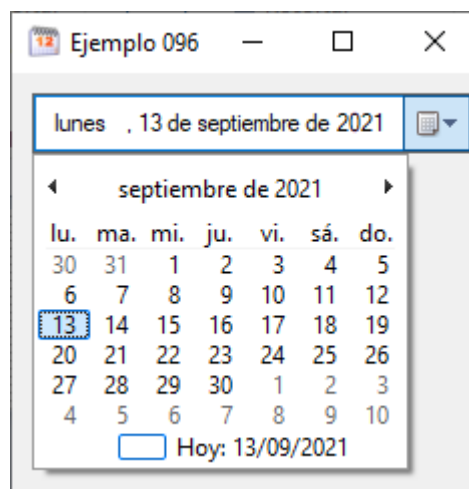
Por último, también disponemos de un mensaje para aplicar un tema de Windows al control ComboBoxEx: [CBEM\\_SETWINDOWTHEME](#).

Para más información sobre los temas de Windows, visita este [enlace](#).

## Ejemplo 95

# Capítulo 55 Control de selección de fecha y hora

Un control de selección de fecha y hora sirve, evidentemente, para que el usuario pueda introducir en una aplicación valores de fechas u horas válidos. Estos controles nos facilitan la vida, ya que, por lo que respecta a fechas, nos proporciona una forma sencilla de validarlas y acceder a calendarios. Y en lo que respecta a horas, facilita la validación de datos.



Ejemplo de control Fecha y hora

Como en todos los controles comunes que estamos viendo, hay que asegurarse de que la DLL ha sido cargada invocando a la función [InitCommonControlsEx](#) indicando el valor de bandera `ICC_DATE_CLASSES` en el miembro `dw/CC` de la estructura [INITCOMMONCONTROLSEX](#) que pasaremos como parámetro.

```
INITCOMMONCONTROLSEX icCE;  
...  
icCE.dwSize = sizeof(INITCOMMONCONTROLSEX);  
icCE.dwICC = ICC_DATE_CLASSES;  
InitCommonControlsEx(&icCE);
```

## Insertar durante la ejecución

Los controles de selección de fecha y hora también se pueden insertar durante la ejecución. Tan sólo hay que crear una ventana de la clase "DATETIMEPICK\_CLASS". Como de costumbre, para hacerlo usaremos las funciones [CreateWindow](#) o [CreateWindowEx](#).

```
case WM_CREATE:
    hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
    hCtrl = CreateWindowEx(0, DATETIMEPICK_CLASS,
NULL,
        WS_BORDER | WS_CHILD | WS_VISIBLE |
WS_TABSTOP | DTS_LONGDATEFORMAT,
        10, 10, 220, 30,
        hwnd, (HMENU)ID_DATETIME,
        hInstance, NULL);
    break;
```

En el parámetro *hMenu* indicaremos el identificador del control y elegiremos los estilos y dimensiones adecuados para nuestro caso.

No olvidemos que cuando se insertan controles durante la ejecución, la fuente por defecto es system. Si queremos cambiarla habrá que crear una fuente y asignársela al control usando un mensaje [WM\\_SETFONT](#). Y no hay que olvidar liberar estos recursos antes de cerrar la aplicación, usando [DeleteObject](#).

```
static HFONT hFont;
...
case WM_CREATE:
    hFont = CreateFont(18, 0, 0, 0, 300, FALSE,
FALSE, FALSE,
        DEFAULT_CHARSET, OUT_TT_PRECIS,
CLIP_DEFAULT_PRECIS,
        PROOF_QUALITY, DEFAULT_PITCH | FF_ROMAN,
"Times New Roman");
    // Asignamos la fuente a nuestro gusto.
    SendMessage(hCtrl, WM_SETFONT, (WPARAM)hFont,
MAKELPARAM(TRUE, 0));
...
case WM_DESTROY:
```

```
DeleteObject(hFont);  
break;
```

## Desde fichero de recursos

Estos controles también pueden situarse en cuadros de diálogo definidos en un fichero de recursos.

Se usa un control general **CONTROL**, con la clase **DATETIMEPICK\_CLASS**, y los estilos generales y específicos deseados.

```
LANGUAGE LANG_NEUTRAL, SUBLANG_NEUTRAL  
IDD_DIALOG2 DIALOG 0, 0, 242, 47  
STYLE DS_3DLOOK | DS_CENTER | DS_MODALFRAME | DS_SHELLFONT |  
WS_CAPTION | WS_VISIBLE | WS_POPUP | WS_SYSMENU  
CAPTION "Diálogo"  
FONT 8, "Ms Shell Dlg"  
{  
    CONTROL        "", ID_DATETIME, DATETIMEPICK_CLASS,  
WS_TABSTOP | DTS_LONGDATEFORMAT, 11, 6, 146, 17, WS_EX_LEFT  
    PUSHBUTTON     "Cancel", IDCANCEL, 176, 24, 50, 14, 0,  
WS_EX_LEFT  
    DEFPUSHBUTTON  "OK", IDOK, 176, 7, 50, 14, 0,  
WS_EX_LEFT  
}
```

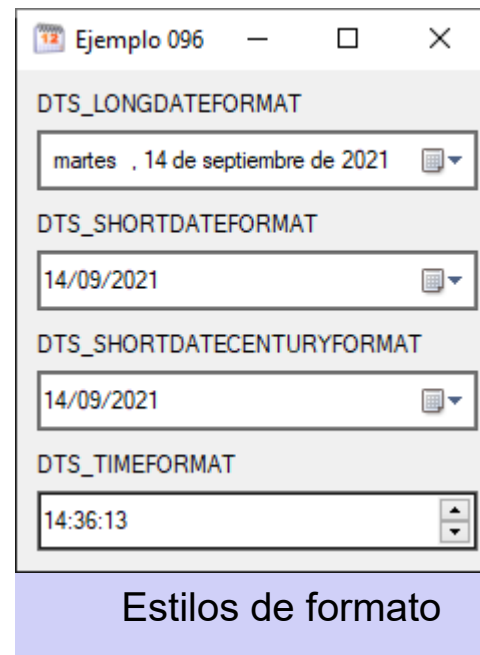
## Estilos

Podemos dividir los **estilos** específicos de estos controles en varios tipos: los de formato, los que afectan a al aspecto, y los que afectan al comportamiento.

Estilos de formato:

- **DTS\_LONGDATEFORMAT**: permite editar fechas expresadas en formato largo.

- **DTS\_SHORTDATEFORMAT:** permite editar fechas expresadas en formato corto, en principio se usarán dos caracteres para el año, aunque esto cambia en función de las opciones de visualización del sistema operativo.
- **DTS\_SHORTDATECENTURYFORMAT:** similar a **DTS\_SHORTDATEFORMAT**, con la diferencia de que los años se muestran con cuatro dígitos.
- **DTS\_TIMEFORMAT:** permite editar horas.

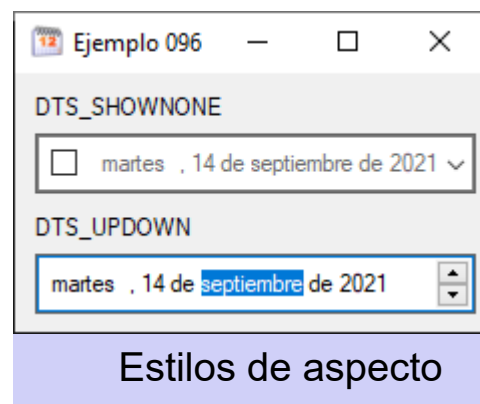


Hay que tener presente que el formato en que se expresan las fechas y horas dependerá de la definición de ciertas opciones locales del sistema. Cambiará el orden en que aparece cada campo (día, mes y año u horas, minutos y segundos), y el idioma de los nombres de días de la semana y de los meses. También varían las horas en función del formato elegido por el usuario en las opciones del sistema operativo, de 12 (PM o AM) ó 24 horas.

Por ejemplo, en el caso de la imagen mostrada no se aprecia diferencia entre los estilos **DTS\_SHORTDATEFORMAT** y **DTS\_SHORTDATECENTURYFORMAT**, y en ambos casos los años se expresan con cuatro dígitos.

Estilos de aspecto:

- **DTS\_SHOWNONE:** permite que opcionalmente el usuario no introduzca una fecha. El control muestra un checkbox, inicialmente marcado, que indica si el control contiene una fecha o no. Si el checkbox no está marcado, el control no contendrá una fecha, y



cualquier intento de leerla no funcionará.

- **DTS\_UPDOWN**: muestra un control updown a la derecha del control. Cuando se trate de horas (estilo **DTS\_TIMEFORMAT**), se mostrará por defecto. Cuando se trate de fechas no se podrá desplegar el control de calendario mensual, pero se podrán modificar los campos (día, mes y año), seleccionándolos y usando el control updown para incrementar o decrementar el valor de cada uno de ellos.
- **DTS\_RIGHTALIGN**: hace que el control de calendario mensual se muestre alineado a la derecha del control de fecha y hora, en lugar de a la izquierda, que es el valor por defecto.

Estilo de comportamiento:

- **DTS\_APPCANPARSE**: permite hacer entradas personalizadas en el control de fecha y hora. El usuario puede pulsar la tecla **F2** e introducir un texto. La aplicación recibirá un mensaje de notificación **DTN\_USERSTRING** y tendrá que analizar la entrada del usuario para validarla y convertirla a un valor de fecha válido.

## Asignar un valor

Por defecto, si no se asigna un valor inicial a un control de fecha y hora, se asignará la fecha y hora actual en el momento en que se crea el control. Esta información se almacena en una estructura **SYSTEMTIME**, de modo que aunque el control sólo visualice una fecha o una hora, en realidad contiene toda la información de tiempo de sistema.

Para asignar un valor diferente podemos usar indistintamente el mensaje **DTM\_SETSYSTEMTIME** o la macro **DateTime\_SetSystemtime**.

Si optamos por el mensaje, en **wParam** indicaremos el valor **GDT\_VALID**, si vamos a inicializar el control con una fecha/hora válida o **GDT\_NONE**, si queremos que el control esté en un estado

"sin fecha". Para este segundo caso el control debe tener el estilo `DTS_SHOWNONE`, y como consecuencia se eliminará la marca del checkbox, quedando el texto en gris.

En `IParam` pasaremos un puntero a una estructura `SYSTEMTIME` con el valor de fecha y/o hora a asignar.

Si todo va bien, el mensaje retornará un valor distinto de cero.

Si optamos por la macro, el primer parámetro es el manipulador de ventana del control, el segundo equivale al `wParam` del mensaje y el tercero equivale al `IParam` del mensaje.

```
SYSTEMTIME st;
int v;
...
st.wYear = 2020;
st.wMonth = 1;
st.wDay = 1;
st.wHour = 12;
st.wMinute = 15;
st.wSecond = 0;
st.wMilliseconds = 0;
// Mensaje:
v = SendDlgItemMessage(hwnd, ID_DATETIME,
DTM_SETSYSTEMTIME, (WPARAM)GDT_VALID, (LPARAM)&st);
// Macro
v = DateTime_SetSystemtime(GetDlgItem(hwnd,
ID_DATETIME), GDT_VALID, &st);
```

## Obtener un valor

Para obtener el valor actual de un control de fecha y hora, siempre que no tenga el estilo `DTS_SHOWNONE` y el checkbox esté sin marcar, podemos usar el mensaje `DTM_GETSYSTEMTIME` o bien la macro `DateTime_GetSystemtime`.

En el caso del mensaje tan sólo hay que pasar en `IParam` un puntero a una estructura `SYSTEMTIME` que recibirá el valor actual del control.



Con la macro el primer parámetro será el manipulador de ventana del control, y el segundo un puntero a una estructura **SYSTEMTIME** donde se situará el valor recuperado del control.

En ambos casos el valor de retorno será **GDT\_VALID** si tiene éxito o **GDT\_NONE** si el control tiene el estilo **DTS\_SHOWNONE** y el checkbox no está marcado, en ese caso el valor de retorno no será válido.

```
SYSTEMTIME st;
int v;
...
v = SendDlgItemMessage(hwnd, ID_DATETIME,
DTM_GETSYSTEMTIME, 0, (LPARAM)&st);
v = DateTime_GetSystemtime(GetDlgItem(hwnd,
ID_DATETIME), (LPARAM)&st);
```

## Establecer rangos

Es posible limitar el rango de fechas que el usuario puede seleccionar en el control. Para ello disponemos del mensaje **DTM\_SETRANGE** y de la macro **DateTime\_SetRange**, que podemos usar indistintamente.

En el caso del mensaje tendremos que pasar en **wParam** una combinación de los valores **GDTR\_MIN** y **GDTR\_MAX**, para indicar si queremos establecer el margen mínimo, el máximo o ambos. En **lParam** pasaremos un puntero a un array de dos estructuras **SYSTEMTIME**, en el que el primer elemento será el margen mínimo y el segundo el máximo del rango.

En el caso de la macro, el primer parámetro será un manipulador de ventana del control, el segundo una combinación de los valores **GDTR\_MIN** y **GDTR\_MAX** y el tercero un array de estructuras **SYSTEMTIME** que definen el rango.

En ambos casos el valor de retorno será distinto de cero si la asignación de rango es establecida, y cero si no es así.

Si no se especifica alguna de las constantes GDTR\_MIN o GDTR\_MAX, ese extremo del rango no se establecerá, y no será necesario que el valor de [SYSTEMTIME](#) correspondiente tenga un valor válido.

En este ejemplo se establece un rango de fechas válidas entre el 15 de noviembre de 2020 y el 15 de diciembre de 2020.

```
SYSTEMTIME rango[2];
int v;
...
rango[0].wYear = 2020;
rango[0].wMonth = 11;
rango[0].wDay = 15;
rango[1].wYear = 2020;
rango[1].wMonth = 12;
rango[1].wDay = 15;
v = SendDlgItemMessage(hwnd, ID_DATETIME, DTM_SETRANGE,
(WPARAM) (GDTR_MIN | GDTR_MAX), (LPARAM) rango);
v = DateTime_SetSystemtime(GetDlgItem(hwnd,
ID_DATETIME), GDTR_MIN | GDTR_MAX, rango);
```

## Obtener rangos

De manera similar podemos recuperar el rango establecido para un control mediante un mensaje [DTM\\_GETRANGE](#) o usando la macro [DateTime\\_GetRange](#).

En el caso del mensaje pasaremos en lParam la dirección de un array de dos estructuras [SYSTEMTIME](#) que recibirá los valores mínimo y máximo del rango.

Para la macro, el primer parámetro es un manipulador de ventana del control, y el segundo la dirección del array.

En ambos casos el valor de retorno será una combinación de los valores GDTR\_MIN y GDTR\_MAX, que nos indicará qué valores del array son válidos.

```
SYSTEMTIME rango[2];
int v;
...
v = SendDlgItemMessage(hwnd, ID_DATETIME, DTM_GETRANGE,
0, (LPARAM)rango);
v = DateTime_GetSystemtime(GetDlgItem(hwnd,
ID_DATETIME), rango);
```

## Atributos del calendario mensual

En el próximo capítulo veremos más detalles sobre el control de calendario mensual. Por ahora nos centraremos en algunas opciones de las que disponemos para personalizar este control que se muestra cuando seleccionamos fechas desplegando el calendario.

Algunas de estas opciones sólo estarán disponibles si no activamos los temas visuales, es decir, si no incluimos en el manifiesto la línea:

```
<assemblyIdentity type="win32"
name="Microsoft.Windows.Common-Controls" version="6.0.0.0"
processorArchitecture="*"
publicKeyToken="6595b64144ccf1df" language="*" />
```

En caso contrario las modificaciones de colores y fuentes no tendrán ningún efecto.

El aspecto gráfico y el comportamiento del control de fecha y hora es bastante diferente si usamos o no los temas visuales.

## Obtener el manipulador de ventana

El control de calendario mensual se crea en el momento en que el usuario pulsa el botón de despliegue, y se destruye cuando el usuario selecciona un valor de fecha. Esto quiere decir que no existe

un manipulador de ventana de este control dentro del control de selección de fecha y hora, de modo que tendremos que obtener ese manipulador cada vez que lo necesitemos.

Para ello podemos procesar el código de notificación `DTN_DROPDOWN` que la aplicación recibirá a través de un mensaje `WM_NOTIFY`.

Si tenemos que realizar alguna tarea cuando el control de calendario mensual sea destruido, por ejemplo, liberar algún recurso, podemos usar el código de notificación `DTN_CLOSEUP`.

Para obtener el manipulador de la ventana de calendario mensual podemos usar el mensaje `DTM_GETMONTHCAL` o la macro `DateTime_GetMonthCal`.

En el caso del mensaje no se requieren parámetros. Para la macro tan sólo es necesario indicar el manipulador de ventana del control de selección de fecha y hora.

```
NMHDR* pnmHdr;
HWND hCMwnd;
...
case WM_NOTIFY:
    pnmHdr = (NMHDR*)lParam;
    switch(pnmHdr->code) {
        case DTN_DROPDOWN:
            hCMwnd = SendMessage(pnmHdr->hwndFrom,
DTM_GETMONTHCAL, 0, 0);
            hCMwnd = DateTime_GetMonthCal(pnmHdr-
>hwndFrom);

            printf("Abierto %d\n", hCMwnd);
            break;
        case DTN_CLOSEUP:
            printf("Cerrado\n");
            break;
    }
    break;
```

## Cambiar la fuente

Los mensajes `DTP_*MC*` hacen referencia al control hijo calendario mensual, pero no se deben enviar a ese control, sino al propio control de selección de fecha y hora. Es decir las opciones que modifiquemos (colores, fuentes o estilos) para el control hijo de calendario mensual tendrán efecto cada vez que se despliegue ese control, ya que se almacenarán con las propiedades del control padre.

Podemos modificar la fuente usada por el control de calendario mensual hijo mediante el mensaje `DTM_SETMCFONT` o la macro `DateTime_SetMonthCalFont`.

Cuando se envíe el mensaje el `wParam` pasaremos un manipulador de la fuente a usar y en `lParam` un valor `BOOL` que indicará si el control debe ser redibujado. Cuando usemos la macro, el primer parámetro será el manipulador de ventana del control de selección de fecha y hora, y los otros dos parámetros son los mismos que para el mensaje.

```
SendDlgItemMessage(hwnd, ID_DATETIME, DTM_SETMCFONT,  
(WPARAM)hFont, (LPARAM)TRUE);  
DateTime_SetMonthCalFont(GetDlgItem(hwnd, ID_DATETIME),  
hFont, TRUE)
```

También podemos recuperar la fuente que se usará para mostrar el control de calendario mensual mediante el envío de un mensaje `DTM_GETMCFONT` o usando la macro `DateTime_GetMonthCalFont`.

El mensaje no requiere parámetros, y la macro sólo necesita que pasemos un manipulador de ventana del control de selección de fecha y hora.

## Cambio de estilos

Disponemos de varias opciones de [estilos de control de calendario mensual](#), por ejemplo, mostrar los números de las

semanas, ocultar la fecha actual de la parte inferior, no mostrar los días del mes anterior y siguiente del actual, etc.

Para asignar los estilos del control de calendario mensual de un control de selección de fecha y hora disponemos del mensaje [DTM\\_SETMCSTYLE](#) y de la macro [DateTime\\_SetMonthCalStyle](#).

Si usamos el mensaje indicaremos el IParam el estilo o combinación de estilos que queremos asignar. Si usamos la macro, el primer parámetro será el manipulador de ventana del control de selección de fecha y hora y el segundo los estilos a asignar.

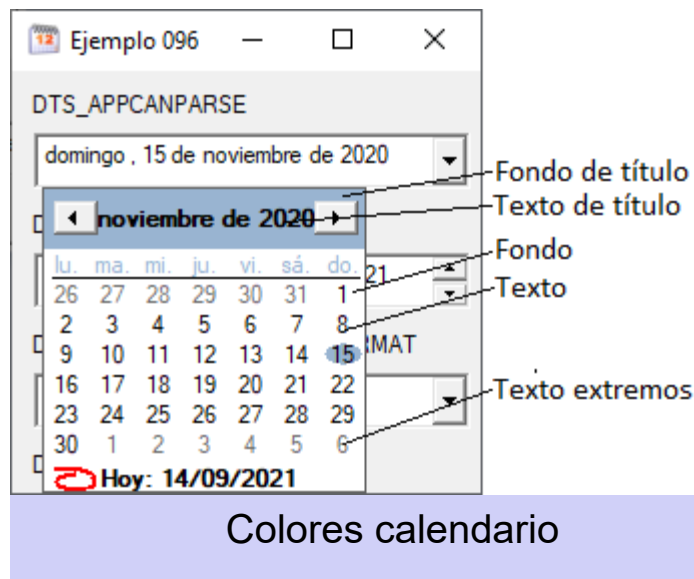
Para recuperar el valor de los estilos actuales del control de calendario mensual asociado a un control de selección de fecha y hora podemos usar el mensaje [DTM\\_GETMCSTYLE](#) o la macro [DateTime\\_GetMonthCalStyle](#).

El mensaje no requiere ningún parámetro, y la macro sólo un manipulador de control de selección de fecha y hora.

## Cambio de colores

Sólo si no están activos los estilos visuales, podemos modificar los colores del control de calendario mensual. Para ello disponemos del mensaje [DTM\\_SETMCCOLOR](#) o de la macro [DateTime\\_SetMonthCalColor](#).

Si se usa el mensaje, en wParam indicaremos de qué parte del control queremos cambiar el color, y en IParam el color que vamos a asignar, en formato [COLORREF](#).



En el caso de la macro, el primer parámetro es un manipulador del control de selección de fecha y hora, y los dos siguientes son los mismos que en el mensaje.

Para indicar qué parte del control queremos modificar existen varias constantes:

- **MCSC\_BACKGROUND**: color de fondo entre meses. Esta parte no se muestra en controles con un único mes, y no tiene efecto en controles de selección de fecha y hora.
- **MCSC\_MONTHBK**: corresponde a la zona 'fondo' de la imagen. El color de fondo de la zona donde se muestran los días.
- **MCSC\_TEXT**: corresponde a la zona 'texto' de la imagen. El color del texto de los días del mes actual.
- **MCSC\_TRAILINGTEXT**: corresponde a la zona 'Texto extremos'. El color del texto de los días correspondientes al mes anterior y siguiente al actual.
- **MCSC\_TITLEBK**: corresponde a la zona 'Fondo de título'.
- **MCSC\_TITLETEXT** corresponde a la zona 'Texto de título'.

Para recuperar el color actualmente asignado a una de estas zonas se puede usar el mensaje [DTM\\_GETMCCOLOR](#) o la macro [DateTime\\_GetMonthCalColor](#).

Si se usa el mensaje, indicaremos en wParam la constante correspondiente a la zona cuyo color queremos recuperar. En el caso de la macro, el primer parámetros será un manipulador de control de selección de fecha y hora y el segundo la constante de la zona.

## **Cerrar calendario**

Podemos cerrar el control de calendario mensual enviando un mensaje [DTM\\_CLOSEMONTHCAL](#) o usando la macro [DateTime\\_CloseMonthCal](#).

El mensaje no necesita parámetros, y la macro sólo un manipulador de ventana del control de selección de fecha y hora.

Este mensaje hará que se cierre el calendario, sin necesidad de que el usuario haya elegido una fecha, y se envíe un mensaje de notificación [DTN\\_CLOSEUP](#) a la ventana propietaria del control de selección de fecha y hora.

## Asignar formato

En este contexto, el formato se refiere a la cadena que se muestra cuando se elige el estilo de formato `DTS_LONGDATEFORMAT`. Si el sistema operativo está en español, por defecto tiene la forma "[dia de semana], [día del mes] de [nombre del mes] de [año]", pero esto puede cambiarse usando el mensaje [DTM\\_SETFORMAT](#) o la macro [DateTime\\_SetFormat](#).

Usando el mensaje indicaremos en `IParam` un puntero a la nueva cadena de formato, o `NULL` si queremos restablecer la cadena por defecto. Con la macro el primer parámetro será el manipulador de ventana del control, y el segundo el puntero a la nueva cadena de formato.

Los literales que se quieran insertar en el formato deberán indicarse entre comillas simples. Para los campos de fecha hay ciertas reglas:

- d: Día del mes, uno o dos dígitos.
- dd o d: Día del mes, dos dígitos, si tiene uno se añade un cero inicial.
- ddd: nombre del día de la semana, con dos letras.
- dddd: nombre del día de la semana, completo.
- h: Hora, uno o dos dígitos en formato de 12 horas.
- hh: Hora, con dos dígitos, en formato de 12 horas, se añade un cero inicial si es necesario.
- H: Hora, uno o dos dígitos en formato de 24 horas.
- HH: Hora, con dos dígitos, en formato de 24 horas, se añade un cero inicial si es necesario.
- m: Minuto, con uno o dos dígitos.



- mm: Minuto, con dos dígitos, se añade un cero inicial si es necesario.
- M: Mes, con uno o dos dígitos
- MM: Mes, con dos dígitos, se añade un cero inicial si es necesario.
- MMM: Mes, con tres letras.
- MMMM: Mes, nombre completo.
- t: Una letra para indicar AM/PM en formato de 12 horas.
- tt: Una letra para indicar AM/PM en formato de 12 horas.
- s: Segundo, con uno o dos dígitos.
- ss: Segundo, con dos dígitos, se añade un cero inicial si es necesario.
- yy: Año, con dos dígitos.
- yyyy: Año, con cuatro dígitos.

```
char* cad = "'Fecha: 'hh': 'mm': 'ss ddd dd'/'MMM'/'yyy";
...
SendDlgItemMessage(hwnd, ID_DATETIME, DTM_SETFORMAT, 0,
(LPARAM)cad);
DateTime_SetFormat(GetDlgItem(hwnd, ID_DATETIME), cad);
```

## Calcular el tamaño del control

Debido a que existen varios estilos de formato de este tipo de controles, el tamaño puede ser muy diferente en función del estilo asignado. Se puede calcular el tamaño ideal de uno de estos controles mediante el mensaje [DTM\\_GETIDEALSIZE](#) o la macro [DateTime\\_GetIdealSize](#).

En el caso del mensaje indicaremos en lParam un puntero a una estructura [SIZE](#) que recibirá las dimensiones ideales para el control. Si se usa la macro indicaremos en el primer parámetro un manipulador del control y en el segundo un puntero a la estructura.

El tamaño varía según la fuente asignada al control, de modo que antes de calcularlo deberemos asignar la fuente, y después

mover el control usando la función [MoveWindow](#).

```
    HFONT hFont;
    SIZE size;

    ...
        hFont = CreateFont(-11, 0, 0, 0, 0, FALSE, FALSE,
FALSE, 1, 0, 0, 0, 0, ("Ms Shell Dlg"));

        CreateWindowEx(0, DATETIMEPICK_CLASS, NULL,
            WS_BORDER | WS_CHILD | WS_VISIBLE | WS_TABSTOP |
DTS_LONGDATEFORMAT | DTS_APPCANPARSE,
            10,30,0,0, /* La anchura y altura no son
relevantes */
            hwnd, (HMENU)ID_DATETIME,
            hInstance, NULL);
        SendDlgItemMessage(hwnd, ID_DATETIME, WM_SETFONT,
(WPARAM)hFont, MAKELPARAM(FALSE, 0));

        SendDlgItemMessage(hwnd, ID_DATETIME,
DTM_GETIDEALSIZE, 0, (LPARAM)&size);
        MoveWindow( GetDlgItem(hwnd, ID_DATETIME), 10, 30,
size.cx, size.cy, TRUE);
```

Aunque la documentación afirma que se calcula el tamaño ideal, lo cierto es que he verificado que esto sólo se cumple para la anchura, y que es mejor que la altura se fije a gusto del programador, ya que parece que el su cálculo se ve influenciado por la altura indicada al crear el control.

## Obtener información

Se puede obtener información adicional sobre el control usando el mensaje [DTM\\_GETDATETIMEPICKERINFO](#) o la macro [DateTime\\_GetDateTimePickerInfo](#).

En el mensaje pasaremos un puntero a una estructura [DATETIMEPICKERINFO](#) en lParam. En la macro pasaremos en el primer parámetro un manipulador de ventana del control de fecha y hora y como segundo parámetro un puntero a la estructura.

Antes de enviar el mensaje o usar la macro hay que inicializar el miembro `cbSize` de la estructura con el valor **`sizeof(DATETIMEPICKERINFO)`**.

La estructura se devolverá con datos relativos al control, el área en pantalla y el estado de la caja de checkbox, el área y estado del botón de despliegue, y los manipuladores de ventana de los controles hijo de edición, up-down y cuadrícula desplegable.

## Códigos de notificación

Existen algunos códigos de notificación específicos para este tipo de controles. Como todos los códigos de notificación, se envían a través de un mensaje [`WM\_NOTIFY`](#).

### Notificación de cambio de fecha y hora

Cuando se produce un cambio en el valor del control se envía a la aplicación un código de notificación [`DTN\_DATETIMECHANGE`](#). En `lParam` se recibe un puntero a una estructura [`NMDATETIMECHANGE`](#) que contiene información sobre el cambio que se haya producido.

La estructura contiene una estructura [`NMHDR`](#), un miembro de banderas que indican el estado del control, siempre que tenga asignado el estilo [`DTS\_SHOWNONE`](#), que indicará si el estado del control es "no date", es decir, si el checkbox está sin marcar con el valor `GDT_NONE`, o si la fecha es válida, y el checkbox marcado, con el valor `GDT_VALID`.

El tercer campo es una estructura [`SYSTEMTIME`](#) con el valor actual del control.

### Control de calendario mensual desplegado

Cada vez que el control de calendario mensual se despliegue se envía a la aplicación un código de notificación [`DTN\_DROPDOWN`](#), y

cuando se cierre un código [DTN\\_CLOSEUP](#).

En ambos casos recibiremos en IParam un puntero a una estructura [NMHDR](#).

## **Campos de retrollamada**

Cuando definimos un formato de fecha usando el mensaje [DTM\\_SETFORMAT](#) o la macro [DateTime\\_SetFormat](#), además de los campos con datos de la fecha, y literales entre comillas sencillas, podemos añadir campos personalizables. Estos son los campos de retrollamada (callback fields). Cuando el control tiene que mostrar la cadena y se encuentra con uno de estos campos envía un código de notificación [DTN\\_FORMAT](#) o [DTN\\_FORMATQUERY](#).

El código de notificación [DTN\\_FORMATQUERY](#) se envía cuando el control tiene que calcular su tamaño ideal. De modo que si el control tiene el estilo `DTS_APPCANPARSE`, la cadena de formato tiene campos de retrollamada y enviamos el mensaje [DTM\\_GETIDEALSIZE](#) recibiremos uno de estos códigos de notificación por cada campo de retrollamada existente.

El código de notificación recibirá en IParam un puntero a una estructura [NMDATETIMEFORMATQUERY](#), y la aplicación debe devolver en el miembro `szMax` el tamaño máximo de la cadena que se podrá usar para ese campo de retrollamada en concreto.

Esto hará posible que el control pueda calcular el tamaño máximo ideal para el control.

Para calcular ese tamaño podemos usar la función [GetTextExtentPoint32](#), usando como manipulador de ventana el del control.

El código de notificación [DTN\\_FORMAT](#) se envía cuando el control tiene que mostrar la cadena en un control con el estilo `DTS_APPCANPARSE` y con campos de retrollamada. Se enviará uno de estos códigos para cada campo de retrollamada.

El programa que procesa este código recibirá en IParam un puntero a una estructura [NMDATETIMEFORMAT](#) y deberá copiar en

el miembro *szDisplay* el texto que debe mostrar el control.

En *pszFormat* estará la subcadena del campo de retrollamada, y en *st* la fecha y hora actualmente almacenadas en el control. Con estos datos deberíamos poder determinar el valor de la cadena.

Los campos de retrollamada se definen mediante uno o más caracteres 'X'. Si necesitamos dos de esos campos podemos usar las subcadenas 'X' y 'XX', o 'XX' y 'XXX', etc.

El usuario puede modificar una fecha en un control de selección de fecha y hora accediendo a uno de los campos individuales del control (año, mes, día, hora, etc). Pero esto también incluye los campos de retrollamada.

El código de notificación [DTN\\_WMKEYDOWN](#) se envía a la aplicación cuando el usuario escribe en uno de los campos de retrollamada.

En *IParam* se recibe un puntero a una estructura [NMDATETIMEWMKEYDOWN](#).

El miembro *nVirtKey* contiene el código de la tecla virtual pulsada. El miembro *pszFormat* identifica el campo de retrollamada mediante su subcadena, y el miembro *st* contiene el valor actual de la fecha y hora almacenada en el control.

```
// Este ejemplo muestra una cadena del tipo "Fecha: mi.  
01/01/2020 02 noche" para el control  
// Dependiendo el valor de la hora, el texto del final será  
mañana (de 6 a 15), tarde  
// (de 15 a 23), o noche (de 23 a 6).
```

```
void CalculaTamanoTexto(HWND hctrl, char* szcad, SIZE*  
lpsize);  
...  
HFONT hFont;  
char* cad = "XX ddd dd'/'MM'/'yyy HH XXX";  
SIZE size;  
NMDATETIMEFORMATQUERY* pnmdbuf;  
...  
    // Crear fuente:  
    hFont = CreateFont(-11, 0, 0, 0, 0, FALSE, FALSE, FALSE,  
1, 0, 0, 0, 0, ("Ms Shell Dlg"));
```

```

// Insertar control:
CreateWindowEx(0, DATETIMEPICK_CLASS, NULL,
    WS_BORDER | WS_CHILD | WS_VISIBLE | WS_TABSTOP |
DTS_LONGDATEFORMAT | DTS_APPCANPARSE,
    10, 30, 2500, 55,
    hwnd, (HMENU)ID_DATETIME,
    hInstance, NULL);
// Asignar la cadena de formato:
DateTime_SetFormat(GetDlgItem(hwnd, ID_DATETIME), cad);
// Calcular tamaño ideal:
SendDlgItemMessage(hwnd, ID_DATETIME, DTM_GETIDEALSIZE,
0, (LPARAM)&size);
// Redimensionar control:
MoveWindow( GetDlgItem(hwnd, ID_DATETIME), 10, 30,
size.cx, 30, TRUE);
...
case WM_NOTIFY:
    switch(pnmHdr->code) {
        case DTN_FORMAT:
            pnmddf = (NMDATETIMEFORMAT*)lParam;
            if(!strcmp(pnmddf->pszFormat, "XX")) {
                strcpy(pnmddf->szDisplay, "Fecha:");
            }
            if(!strcmp(pnmddf->pszFormat, "XXX")) {
                if(pnmddf->st.wHour >= 6 && pnmddf-
>st.wHour < 15)
                    strcpy(pnmddf->szDisplay, "mañana");
                else if(pnmddf->st.wHour >= 15 &&
pnmddf->st.wHour < 23)
                    strcpy(pnmddf->szDisplay, "tarde");
                else strcpy(pnmddf->szDisplay, "noche");
            }
            printf("Format\n");
            return 0;
        case DTN_FORMATQUERY:
            // Obtener tamaños de callback fields:
            pnmddf = (NMDATETIMEFORMATQUERY*)lParam;
            if(!strcmp(pnmddf->pszFormat, "XX")) { //
Valor fijo:
                CalculaTamanoTexto(GetDlgItem(hwnd,
ID_DATETIME), "Fecha:", &pnmddf->szMax);
            }
            if(!strcmp(pnmddf->pszFormat, "XXX")) { //
Mañana, tarde o noche, en función de hora.
                CalculaTamanoTexto(GetDlgItem(hwnd,
ID_DATETIME), "mañana", &pnmddf->szMax);
            }

```

```

        return 0;
    case DTN_WMKEYDOWN:
        pnmvtk = (NMDATETIMEWMKEYDOWN*)lParam;
        if(!strcmp(pnmvtk->pszFormat, "XXX")) {
            if(pnmvtk->nVirtKey == 'M') {
                pnmvtk->st.wHour=6;
            }
            if(pnmvtk->nVirtKey == 'T') {
                pnmvtk->st.wHour=15;
            }
            if(pnmvtk->nVirtKey == 'N') {
                pnmvtk->st.wHour=23;
            }
        }
        return 0;
    ...
void CalculaTamanoTexto(HWND hctrl, char* szcad, SIZE*
lpsize) {
    HDC hdc;

    hdc = GetDC(hctrl);
    GetTextExtentPoint32(hdc, szcad, strlen(szcad), lpsize);
    ReleaseDC(hctrl, hdc);
}

```

En rigor, para calcular el tamaño del campo 'XXX' tendríamos que haber usado el mayor tamaño para las tres cadenas, 'mañana', 'tarde', y 'noche'. Pero asumimos que la de 'mañana' será la más larga al tener más caracteres.

## Cadenas de usuario

Si el control tiene el estilo DTS\_APPCANPARSE, el usuario podrá escribir texto en el control de edición asociado al control de selección de fecha y hora. Podrá, por ejemplo, introducir fechas incompletas "12/08", y que la aplicación calcule los campos faltantes usando valores por defecto; podrá introducir fechas como "ayer" o "mañana", etc.

Si el usuario pulsa **F2** se seleccionará todo el texto del control y se entrará en edición.

La aplicación deberá procesar estas cadenas y realizar las comprobaciones y cálculos necesarios para obtener una fecha válida.

Para ello, la aplicación recibirá un código de notificación [DTN\\_USERSTRING](#) cuando el usuario finalice la edición del texto. En `LPARAM` recibirá un puntero a una estructura [NMDATETIMESTRING](#) en la que el miembro *pszUserString* contendrá la cadena introducida por el usuario, *st* el valor de fecha y hora actualmente almacenadas en el control, y en *dwflags* el valor del checkbox si además el control tiene el estilo `DTS_SHOWNONE`, que será `GDT_VALID` si el checkbox está marcado o `GDT_NONE` en caso contrario.

```
case DTN_USERSTRING:
    pnmmts = (NMDATETIMESTRING*)lParam;
    if(!strcmp(pnmmts->pszUserString, "hoy"))
        GetLocalTime(&(pnmmts->st));
    else if(!strcmp(pnmmts->pszUserString,
"mañana")) {
        GetLocalTime(&(pnmmts->st));
        pnmmts->st.wDay++;
    } else if(!strcmp(pnmmts->pszUserString,
"ayer")) {
        GetLocalTime(&(pnmmts->st));
        pnmmts->st.wDay--;
    }
    return 0;
```

La función [GetLocalTime](#) sirve para obtener la fecha y hora local actual. Para obtener la fecha y hora en formato (UTC) se usa la función [GetSystemTime](#).

## Foco del teclado

Por último, cuando el control pierde el foco del teclado, la aplicación recibe un código de notificación [NM\\_KILLFOCUS](#).



Cuando el control recibe el foco del teclado, la aplicación recibe un código de notificación `NM_SETFOCUS`.

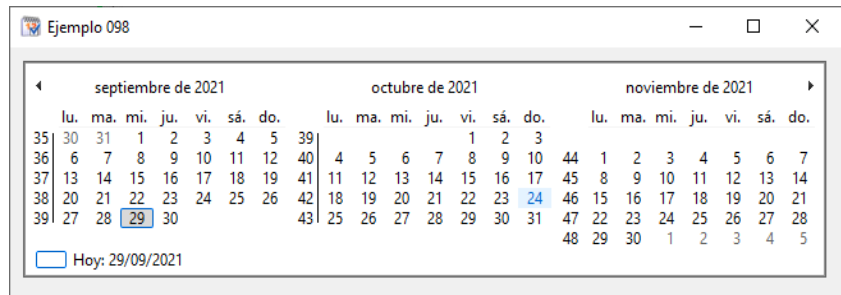
## **Ejemplo 96**

## **Ejemplo 97**

# Capítulo 56 Control de calendario

El control de calendario mensual permite elegir fechas mediante la selección desde un calendario que muestra uno o más meses. Estos controles proporcionan una forma sencilla de seleccionar fechas o intervalos de fechas.

Como en todos los controles comunes que estamos viendo, hay que asegurarse de que la DLL ha sido cargada invocando a la función [InitCommonControlsEx](#) indicando el valor de bandera `ICC_DATE_CLASSES` en el miembro `dwICC` de la estructura [INITCOMMONCONTROLSEX](#) que pasaremos como parámetro.



Ejemplo de control calendario mensual

```
INITCOMMONCONTROLSEX icce;
...
icce.dwSize = sizeof(INITCOMMONCONTROLSEX);
icce.dwICC = ICC_DATE_CLASSES;
InitCommonControlsEx(&icce);
```

## Insertar durante la ejecución

Como todos los controles que hemos visto hasta ahora, los de calendario también se pueden insertar durante la ejecución. Tan sólo hay que crear una ventana de la clase "MONTHCAL\_CLASS". Para hacerlo usaremos las funciones [CreateWindow](#) o [CreateWindowEx](#).

```
case WM_CREATE:
    hInstance = ((LPCREATESTRUCT)lParam)->hInstance;
    CreateWindowEx(0, MONTHCAL_CLASS, NULL,
        WS_BORDER | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
        10, 10, 200, 200,
        hwnd, (HMENU)ID_CALENDAR1,
        hInstance, NULL);
    break;
```

En el parámetro `hMenu` indicaremos el identificador del control y elegiremos los estilos y dimensiones adecuados para nuestro caso.

Si queremos cambiar la fuente usada para mostrar el control, habrá que crear una fuente y asignársela usando un mensaje [WM\\_SETFONT](#). Y no hay que olvidar liberar estos recursos antes de cerrar la aplicación, usando [DeleteObject](#).

```
static HFONT hFont;
...
case WM_CREATE:
...
    hFont = CreateFont(18, 0, 0, 0, 300, FALSE, FALSE, FALSE,
        DEFAULT_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
        PROOF_QUALITY, DEFAULT_PITCH | FF_ROMAN, "Times New Roman");
```

```

        // Asignamos la fuente a nuestro gusto.
        SendMessage(hCtrl, WM_SETFONT, (WPARAM)hFont, MAKELPARAM(TRUE, 0));
...
    case WM_DESTROY:
        DeleteObject(hFont);
        break;

```

Un control de calendario mensual puede mostrar uno o más meses. El número de meses que se mostrarán dependerá de las dimensiones del control, tanto en anchura como en altura.

En la imagen del ejemplo se muestran tres meses en sentido horizontal, pero si las dimensiones del control lo permiten se podrían mostrar en vertical o en una cuadrícula con varias columnas y filas.

Disponemos de dos mensajes para ayudarnos a calcular las dimensiones del control:

- **MCM\_GETMINREQRECT**: calcula las dimensiones mínimas para mostrar un único mes en el control. En lParam tenemos que pasar un puntero a una estructura **RECT** que recibirá el tamaño mínimo del control.
- **MCM\_SIZERECTTOMIN**: calcula las dimensiones mínimas para mostrar varios meses en el control. En este caso deberemos pasar en lParam un puntero a una estructura **RECT** con las dimensiones aproximadas a la entrada, y nos devolverá, en la misma estructura, las dimensiones corregidas para mostrar los meses que caben en el rectángulo de entrada.

En el primer caso, los miembros *left* y *top* serán cero, y en *right* y *bottom* se devolverá la anchura y altura del control, respectivamente.

En el segundo caso la anchura disponible se calcula como *right-left* y la altura como *bottom-top*, y al retornar, *left* y *top* serán cero, y *right* y *bottom* serán la anchura y altura, respectivamente.

El mensaje **MCM\_SIZERECTTOMIN** sólo funciona si están activos los estilos visuales, usando el manifiesto adecuado.

En ambos casos deberemos usar la función **MoveWindow** para redimensionar el control.

Ajustar el tamaño del control para mostrar un mes:

```

// Ejemplo para mostrar un mes:
RECT re;
...
    SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_GETMINREQRECT, 0, (LPARAM)&re);
    MoveWindow( GetDlgItem(hwnd, ID_CALENDAR1), 10, 10, re.right, re.bottom, TRUE);

```

Ajustar el tamaño del control para mostrar varios meses:

```

// Ejemplo para mostrar varios meses:
RECT re;
...
    re.top = 10;
    re.left = 20;
    re.right = 500;
    re.bottom = 200;
    SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_GETMINREQRECT, 0, (LPARAM)&re);
    MoveWindow( GetDlgItem(hwnd, ID_CALENDAR1), 40, 40, re.right, re.bottom, TRUE);

```

Alternativamente se pueden usar las macros **MonthCal\_SizeRectToMin** y **MonthCal\_GetMinReqRect** en lugar de **MCM\_SIZERECTTOMIN** y **MCM\_GETMINREQRECT**, respectivamente.

## Desde fichero de recursos

También podemos insertar controles de calendario en nuestros cuadros de diálogo usando ficheros de recursos.

Esto nos facilita las cosas, sobre todo si diseñamos nuestros recursos usando un editor.

Se usa un control general [CONTROL](#), con la clase `MONTHCAL_CLASS`, y los estilos generales y específicos que queramos aplicar.

```
LANGUAGE LANG_NEUTRAL, SUBLANG_NEUTRAL
IDD_DIALOG1 DIALOG 0, 0, 331, 225
STYLE DS_3DLOOK | DS_CENTER | DS_MODALFRAME | DS_SHELLFONT | WS_CAPTION | WS_VISIBLE |
WS_POPUP | WS_SYSMENU
CAPTION "Dialog"
FONT 8, "Ms Shell Dlg"
{
    CONTROL            "", 0, MONTHCAL_CLASS, WS_TABSTOP | MCS_NOTODAY, 13, 10, 250, 202,
WS_EX_LEFT
    DEFPUSHBUTTON      "OK", IDOK, 269, 10, 50, 14, 0, WS_EX_LEFT
    PUSHBUTTON         "Cancel", IDCANCEL, 269, 27, 50, 14, 0, WS_EX_LEFT
}
```

## Estilos

Disponemos de algunos [estilos](#) específicos para estos controles que permiten personalizar su aspecto y comportamiento.

- `MCS_DAYSTATE`: permite mostrar algunas fechas en negrita, para ello el control enviará códigos de notificación [MCN\\_GETDAYSTATE](#) para solicitar a la aplicación la información necesaria.
- `MCS_MULTISELECT`: permite que el usuario pueda seleccionar un rango de fechas.
- `MCS_WEEKNUMBERS`: muestra a la izquierda de cada fila de cada calendario el número de la semana dentro del año.
- `MCS_NOTODAYCIRCLE`: oculta el resaltado del día actual en el calendario correspondiente.
- `MCS_NOTODAY`: oculta la leyenda al pie del calendario con la fecha del día actual. Esta leyenda actúa, cuando es visible, como un botón que selecciona la fecha actual en el control.
- `MCS_NOTRAILINGDATES`: oculta las fechas de los primeros días de la primera semana correspondientes al mes anterior al primero mostrado y las de los últimos días de la última semana correspondientes a la mes siguiente al último mostrado.
- `MCS_SHORTDAYSOFWEEK`: las leyendas para los días de la semana se muestran con una letra (L, M, X, etc) en lugar de usar tres caracteres (lu., ma., mi., etc).
- `MCS_NOSELCHANGEONNAV`: si el usuario ha seleccionado alguna fecha o un rango de fechas, la selección no cambia si al navegar a otros meses desaparece del rango de meses mostrado en el control. De este modo el usuario puede seleccionar más fechas de las que son visibles en el control.

## Control de calendario de selección simple

Por defecto, salvo que se especifique el estilo [MCS\\_MULTISELECT](#), los controles de calendario sólo permiten seleccionar una fecha.

### Seleccionar fecha

Para cambiar la fecha seleccionada en un control de calendario podemos usar el mensaje [MCM\\_SETCURSEL](#), indicando el lParam un puntero a una estructura [SYSTEMTIME](#) con la fecha a seleccionar. Sólo se usaran los miembros relativos a la fecha, y el resto se ignoran.

```
SYSTEMTIME fecha;
...
fecha.wDay = 23;
```

```
fecha.wMonth = 3;
fecha.wYear = 2023;
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_SETCURSEL, 0, (LPARAM)&fecha);
```

La macro [MonthCal\\_SetCurSel](#) es equivalente al mensaje [MCM\\_SETCURSEL](#).

## Obtener fecha seleccionada

Para obtener el valor de la fecha actualmente seleccionada usaremos el mensaje [MCM\\_GETCURSEL](#), pasando en lParam un puntero a una estructura [SYSTEMTIME](#), que recibirá el valor de la fecha actualmente seleccionada en el control.

```
SYSTEMTIME fecha;
...
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_GETCURSEL, 0, (LPARAM)&fecha);
```

La macro [MonthCal\\_GetCurSel](#) es equivalente.

## Selección múltiple

Si se especifica el estilo [MCS\\_MULTISELECT](#), el usuario podrá seleccionar un rango de fechas. Si no se modifica explícitamente, el rango máximo es una semana.

Si se selecciona más de una fecha, deben ser consecutivas. La selección se hace pulsando con el botón izquierdo sobre una fecha y, manteniendo pulsado el botón, moverse a otra fecha. También se puede seleccionar una fecha y, manteniendo pulsada la tecla `SHIFT`, seleccionar una segunda fecha.

Si entre la primera y la segunda fecha seleccionadas hay más días que el máximo rango permitido, sólo se seleccionará el número máximo de días a partir de la primera selección.

## Asignación de varias fechas

Para asignar un rango de fechas a un control de calendario se usa el mensaje [MCM\\_SETSELRANGE](#), indicando en lParam un puntero a un array de dos elementos que contendrán las fechas mínima y máxima a seleccionar.

El orden en que se indiquen las fechas es indiferente, pero es importante que el rango sea menor o igual al máximo permitido para el control, o el mensaje fallará.

```
SYSTEMTIME fecha[2];
...
fecha[0].wDay = 20;
fecha[0].wMonth = 8;
fecha[0].wYear = 2048;
fecha[1].wDay = 15;
fecha[1].wMonth = 8;
fecha[1].wYear = 2048;
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_SETSELRANGE, 0, (LPARAM)&fecha);
```

La macro [MonthCal\\_SetSelRange](#) es equivalente.

## Obtener asignación múltiple

Para obtener el rango de fechas seleccionadas en el control se usa el mensaje [MCM\\_GETSELRANGE](#), pasando en lParam un puntero a un array de dos elementos que recibirá las fechas que definen el rango.

```
SYSTEMTIME fecha[2];
...
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_GETSELRANGE, 0, (LPARAM)&fecha);
```

También se puede usar la macro equivalente [MonthCal\\_GetSelRange](#).

## Rango máximo de selección

Para modificar el rango máximo de fechas que se pueden seleccionar en un control de calendario usaremos el mensaje [MCM\\_SETMAXSELCOUNT](#), indicando en wParam el nuevo valor máximo del rango.

```
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_SETMAXSELCOUNT, (WPARAM)30, 0);
```

O usar la macro equivalente [MonthCal\\_SetMaxSelCount](#).

Para obtener el valor del rango máximo seleccionable se usa el mensaje [MCM\\_GETMAXSELCOUNT](#), o la macro [MonthCal\\_GetMaxSelCount](#).

## Selección fuera de la vista

Por defecto, si el usuario navega a través del calendario y la selección actual queda fuera de la vista, automáticamente se seleccionará un nuevo rango del mismo tamaño en la vista actual. Esto también se aplica a controles de calendario de selección simple.

Podemos evitar este comportamiento asignando el estilo [MCS\\_NOSELCHANGEONNAV](#). De este modo la selección actual se mantiene aunque no sea visible en el control.

Esta es la única forma de que un usuario pueda seleccionar rangos más grandes de los que permite mostrar el control.

## Fechas seleccionables

Si queremos limitar el rango de fechas que el usuario puede seleccionar disponemos del mensaje [MCM\\_SETRANGE](#). En wParam indicaremos una combinación de los valores GDTR\_MAX y GDTR\_MIN, dependiendo de que límites queramos establecer, y en lParam pasaremos un puntero a un array de dos elementos con las fechas mínima y máxima permitidas.

Si sólo queremos establecer el límite superior usaremos el valor GDTR\_MAX en wParam, y si sólo queremos establecer un límite inferior, el valor GDTR\_MIN. En los dos casos hay que pasar el array con dos elementos, pero sólo se tendrán en cuenta los valores en función del valor de wParam.

```
SYSTEMTIME fecha[2];
...
/* Sólo se permiten fechas entre el 1 de enero y el 31 de diciembre de 2022 */
fecha[0].wDay = 1;
fecha[0].wMonth = 1;
fecha[0].wYear = 2022;
fecha[1].wDay = 31;
fecha[1].wMonth = 12;
fecha[1].wYear = 2022;
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_SETRANGE, (WPARAM)(GDTR_MAX|GDTR_MIN),
(LPARAM)&fecha);
```

La macro [MonthCal\\_SetRange](#) es equivalente.

Para obtener el rango de fechas disponibles en un control de calendario se usa el mensaje [MCM\\_GETRANGE](#), indicando en IParam un array de dos elementos que recibirá las fechas mínima y máxima del rango seleccionable por el usuario.

El valor de retorno será una combinación de los valores GDTR\_MAX y GDTR\_MIN, que indicará qué elementos del array contienen valores válidos. Si el valor de retorno es cero, significa que no se han establecido límites.

También podemos usar la macro [MonthCal\\_GetRange](#) para esta tarea.

## Aspecto gráfico

Disponemos de varias opciones para modificar el aspecto en pantalla de los controles de calendario, tamaños, colores, fuentes, etc.

### Borde

Cuando hablamos del borde de un control de calendario nos referimos al margen entre el límite exterior del control y el conjunto de los calendarios de cada mes que se muestran en el interior. Ese margen rodea a todos los meses.

Podemos establecer el tamaño del borde mediante un mensaje [MCM\\_SETCALENDARBORDER](#). En wParam indicaremos un valor de tipo BOOL, TRUE indicará que queremos establecer un nuevo valor para la anchura del borde en pixels, indicado en IParam, y FALSE para restablecer el valor por defecto.

Es importante establecer el valor del borde antes de calcular el tamaño del control. Al igual que la fuente, el tamaño del borde influye en el cálculo del tamaño del control.

La macro [MonthCal\\_SetCalendarBorder](#) se despliega como un envío de éste mensaje.

```
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_SETCALENDARBORDER, (WPARAM) TRUE, (LPARAM) 30);
```

Para recuperar el valor actual del tamaño del borde se usa el mensaje [MCM\\_GETCALENDARBORDER](#), o la macro equivalente [MonthCal\\_GetCalendarBorder](#).

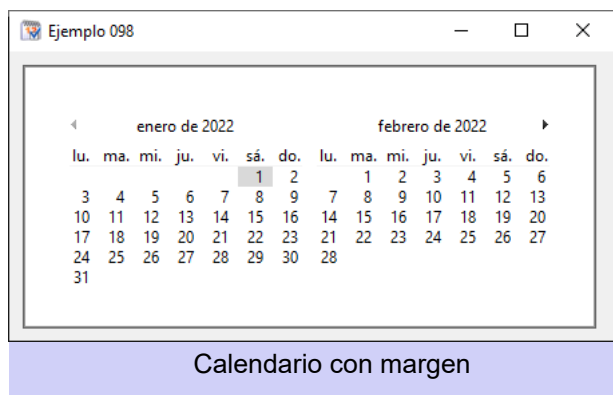
### Colores

Podemos modificar los colores de un control de calendario, pero sólo si no están activos los estilos visuales. Para ello disponemos del mensaje [MCM\\_SETCOLOR](#).

En wParam indicaremos de qué parte del control queremos cambiar el color, y en IParam el color que vamos a asignar, en formato [COLORREF](#).

Para indicar qué parte del control queremos modificar existen varias constantes:

- MCSC\_BACKGROUND: color de fondo de la separación entre meses.
- MCSC\_MONTHBK: corresponde a la zona 'fondo' de la imagen. El color de fondo de la zona donde se muestran los días.
- MCSC\_TEXT: corresponde a la zona 'texto' de la imagen. El color del texto de los días del mes actual.



- MCSC\_TRAILINGTEXT: corresponde a la zona 'Texto extremos'. El color del texto de los días correspondientes al mes anterior y siguiente al actual.
- MCSC\_TITLEBK: corresponde a la zona 'Fondo de título'.
- MCSC\_TITLETEXT corresponde a la zona 'Texto de título'.

Para recuperar el color actualmente asignado a una de estas zonas se puede usar el mensaje [MCM\\_GETCOLOR](#), indicando en wParam la constante correspondiente a la zona cuyo color queremos recuperar.

Alternativamente, también se pueden usar las macros [MonthCal\\_SetColor](#) y [MonthCal\\_GetColor](#), respectivamente.

## Estado de días

Si el estilo [MCS\\_DAYSTATE](#) está activo, podremos especificar un estado resaltado para cada día de los meses mostrados en el control, que se indicará mostrando el texto del día correspondiente en negrita.

Como cada mes tiene como máximo 31 días, y el estado de resalte es un valor binario, se usa un valor de 32 bits para especificar el estado de todos los días de un mes. Más concretamente, se usa un valor de tipo [MONTHDAYSTATE](#). Los bits con valor 1 indicarán que el día correspondiente se deberá mostrar resaltado.

Por otra parte, como un control de calendario puede mostrar varios meses, tendremos que especificar tantas de estas estructuras como meses contenga el control.

Así, usando el mensaje [MCM\\_SETDAYSTATE](#) podemos establecer qué días se mostrarán resaltados en un control de calendario, indicando en lParam la dirección de un array de elementos de tipo [MONTHDAYSTATE](#), uno por cada mes a asignar, y en wParam el número de elementos que contiene el array.

También podemos usar la macro [MonthCal\\_SetDayState](#).

Evidentemente, cada vez que los meses mostrados en el control cambien deberemos asignar los nuevos estados a los meses mostrados.

Pero todo esto es mucho más sencillo si procesamos el código de notificación [MCN\\_GETDAYSTATE](#).

### Nota:

Hay un error en el fichero de cabecera "commctrl.h" que se incluye con MinGW en la definición de este código de notificación.

Donde pone:

```
#define MCN_GETDAYSTATE (MNN_FIRST+3)
```

Debe poner:

```
#define MCN_GETDAYSTATE (MCN_FIRST-1)
```

Siempre que tengamos asignado el estilo [MCS\\_DAYSTATE](#), cada vez que cambien los meses mostrados en el control recibiremos un código de notificación [MCN\\_GETDAYSTATE](#). En lParam tendremos un puntero a una estructura [NMDAYSTATE](#) con toda la información necesaria para actualizar el estado de los meses mostrados en el control.

Como toda las estructuras recibidas en códigos de notificación, esta también contiene en primer lugar una estructura [NMHDR](#) con la información relativa a la notificación (manipulador de ventana, identificador y código de notificación). Además contiene una estructura [SYSTEMTIME](#), con la fecha de comienzo del rango requerido, un puntero a un array de valores [MONTHDAYSTATE](#), que deberemos asignar al procesar el mensaje y un valor entero, con el número de elementos que debe contener ese array.



Por ejemplo, el control de la imagen anterior está mostrando dos meses, enero y febrero de 2022. Enero empieza en sábado, lo que implica que deberán mostrarse los últimos días del mes anterior de esa semana. Esto es así aunque hayamos asignado el estilo MCS\_NOTRILINGDATES, que oculta esos días.

El segundo mes termina en lunes, lo que implica que deberán mostrarse los primeros días del mes siguiente. Por lo tanto el array deberá contener cuatro elementos y la fecha que se solicitará será la del día uno del mes anterior al primero, es decir, el 27 de diciembre de 2021 (el último lunes de diciembre de 2021).

```
/* En este ejemplo supondremos que sólo se pueden seleccionar fechas del año 2022 */
/* El array 'estados' contiene los estados resaltados para los sábados y domingos desde
   diciembre de 2021 a enero de 2023 */
MONTHDAYSTATE estados[] =
{0x3060c18,
 0x3060c183, 0x60c1830, 0x60c1830, 0x20c18306, 0x183060c1, 0x3060c18,
 0x60c18306, 0xc183060, 0x183060c, 0x3060c183, 0x60c1830, 0x4183060c,
 0x183060c1};
LPNMDAYSTATE lpmDS;
...
case WM_NOTIFY:
    lpmDS = (LPNMDAYSTATE)lParam;
    switch(lpmDS->nmhdr.code) {
        case MCN_GETDAYSTATE:
            if(lpmDS->stStart.wYear == 2021)
                lpmDS->prgDayState = &estados[0];
            else
                lpmDS->prgDayState = &estados[lpmDS->stStart.wMonth];
            return 1;
    }
```

Lo cierto es que de la fecha recibida sólo nos interesa el mes y el año, el resto de los datos son irrelevantes.

El día será el correspondiente al primer día de la semana del primer mes, aunque ese día corresponda al mes anterior. El valor de *wDayOfWeek* será válido y nos puede servir para generar los valores de estados automáticamente.

Los bits en la estructura [MONTHDAYSTATE](#) se empiezan a contar a partir de la derecha, el bit menos significativo corresponde al primer día del mes.

Las operaciones de rotación de bits son sencillas, para rotar un bit a la izquierda el valor contenido en un entero basta multiplicar por dos, o podemos usar directamente el operador de rotación de bits. Por ejemplo, si en el ejemplo anterior queremos resaltar el 6 de enero de 2022 podemos usar esta sentencia:

```
estados[1] |= (MONTHDAYSTATE) (0x1 << 5);
```

## Calendarios contenidos

Disponemos de un mensaje para obtener información sobre el número de calendarios. El mensaje [MCM\\_GETCALENDARCOUNT](#) nos devuelve el número de calendarios actualmente mostrados en el control, no es necesario indicar ningún parámetro.

La macro [MonthCal\\_GetCalendarCount](#) es equivalente.

## Obtener información

Por otra parte, el mensaje [MCM\\_GETCALENDARGRIDINFO](#) sirve para obtener información sobre cada una de las zonas que definen un calendario. En *lParam* pasaremos un puntero a una estructura

**MCGRIDINFO** en la que se nos devolverá la información requerida.

Antes de enviar el mensaje deberemos iniciar algunos miembros de la estructura. *cbSize* debe contener el tamaño de la estructura:

```
mcGI.cbSize = sizeof(MCGRIDINFO);
```

El miembro *dwPart* debe contener el valor de la constante que indica qué información en concreto queremos obtener. Puede ser uno de los siguientes valores:

- **MCGIP\_CALENDARCONTROL**: el control completo, que puede contener hasta 12 calendarios.
- **MCGIP\_NEXT**: el botón de navegación "siguiente".
- **MCGIP\_PREV**: el botón de navegación "anterior".
- **MCGIP\_FOOTER**: el pie del calendario.
- **MCGIP\_CALENDAR**: un calendario específico. Se den asignar también los miembros *iCalendar* y *pszName*.
- **MCGIP\_CALENDARHEADER**: la cabecera de calendario. Se den asignar también los miembros *iCalendar* y *pszName*.
- **MCGIP\_CALENDARBODY**: el cuerpo del calendario. Hay que asignar el miembro *iCalendar*.
- **MCGIP\_CALENDARROW**: una fila de calendario determinada por *iCalendar* e *iRow*.
- **MCGIP\_CALENDARCELL**: una celda de calendario dada determinada por *iCalendar*, *iRow*, *iCol*, *bSelected* y *pszName*.

También deberemos indicar un valor para el miembro *dwFlags*, dependiendo de qué información queremos que nos sea retornada. Puede ser una combinación de uno o varios de los valores siguientes:

- **MCGIF\_DATE**: se devolverán los valores de *stStart* y *stEnd*.
- **MCGIF\_RECT**: se devolverá *rc*.
- **MCGIF\_NAME**: se devolverá *pszName*.

Para determinar de qué parte del calendario estamos solicitando la información, habrá que asignar otros campos como *iCalendar*, *iRow*, *iCol*, *bSelected* o *pszName*, dependiendo de cada caso.

En el caso de que se solicite información en forma de cadenas, en *pszName* asignaremos la dirección de un buffer, y el *cchName* el tamaño de ese buffer.

Al retornar, los miembros *bSelected*, *stStart*, *stEnd*, *rc* y la cadena apuntada por *pszName* contendrán la información solicitada, dependiendo en cada caso de los valores de entrada.

```
MCGRIDINFO mcGI;
WCHAR cad[100];
...
    mcGI.cbSize = sizeof(MCGRIDINFO);
    mcGI.dwPart = MCGIP_CALENDAR;
    mcGI.iCalendar = 0;
    mcGI.pszName = cad;
    mcGI.cchName = 100;
    mcGI.dwFlags = MCGIF_NAME;
    SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_GETCALENDARGRIDINFO, 0, (LPARAM)&mcGI);
    /* En mcGI.pszName estará la cadena con la fecha actualmente seleccionada, por
ejemplo
        5 de enero de 2022 */
```

La macro **MonthCal\_GetCalendarGridInfo** es equivalente al mensaje **MCM\_GETCALENDARGRIDINFO**.

Otro dato que podemos solicitar es el de las fechas actualmente mostradas en el control. No confundir con las fechas seleccionables.

## Rangos visibles

El mensaje [MCM\\_GETMONTHRANGE](#) devuelve en lParam los valores de [SYSTEMTIME](#) que determinan el rango de fechas. Si en wParam indicamos el valor GMR\_DAYSTATE se incluirán las fechas de los meses mostrados parcialmente al inicio y al final de cada calendario, si se indica el valor GMR\_VISIBLE sólo se incluirán las fechas de los meses que sean mostrados completamente.

El array de dos fechas que pasemos en lParam debe ser una dirección válida. El sistema no proporciona esa memoria automáticamente.

```
SYSTEMTIME fecha[2];
...
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_GETMONTHRANGE, (WPARAM)GMR_VISIBLE,
(LPARAM) fecha);
```

La macro [MonthCal\\_GetMonthRange](#) es equivalente.

## Medidas de la cadena 'hoy'

Podemos obtener la medida de la anchura máxima para la cadena 'hoy', mostrada al pie del control si no se ha especificado el estilo MCS\_NOTODAY, usando el mensaje [MCM\\_GETMAXTODAYWIDTH](#), sin parámetros.

```
INT x;
...
x = SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_GETMAXTODAYWIDTH, 0, 0);
```

También se puede usar la macro [MonthCal\\_GetMaxTodayWidth](#).

## Tipos de calendario

Existen varios tipos de calendarios, además del calendario gregoriano que usamos generalmente en occidente. En otras partes del mundo se usan calendarios diferentes: Japón, Taiwan, Corea, etc.

El control de calendario dispone de un identificador que determina qué tipo de calendario se está usando. Podemos asignar un nuevo identificador mediante el mensaje [MCM\\_SETCALID](#), indicando en wParam el valor del nuevo identificador. Este valor puede ser uno de los valores definidos para [CALID](#).

Para obtener el identificador de un control de calendario se usa el mensaje [MCM\\_GETCALID](#).

También se pueden usar las macros [MonthCal\\_SetCALID](#) y [MonthCal\\_GetCALID](#), respectivamente.

### Nota:

He intentado hacer algún ejemplo de cambio de ID, pero aparentemente no tiene ningún efecto en la apariencia del control.

Otra opción de la que disponemos es elegir qué día empieza cada semana. En mi configuración de Windows las semanas empiezan en lunes, como se ve en las imágenes de ejemplo, pero esto se puede modificar usando el mensaje [MCM\\_SETFIRSTDAYOFWEEK](#), indicando en lParam qué día de la semana será el primero, empezando en 0 para el lunes.

Si no se modifica, el valor por defecto es LOCALE\_FIRSTDAYOFWEEK, que depende de la configuración regional.

```
/* El primer día de la semana es el domingo */
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_SETFIRSTDAYOFWEEK, 0, (LPARAM)6);
```

El valor de retorno es un DWORD, donde la palabra de mayor peso será TRUE si el valor previo no era LOCALE\_IFIRSTDAYOFWEEK, y la palabra de menor peso contendrá el valor previo para el primer día de la semana.

La macro [MonthCal\\_SetFirstDayOfWeek](#) equivale a ese mensaje.

Para obtener el valor actual para el primer día de la semana de un control de calendario se usa el mensaje [MCM\\_GETFIRSTDAYOFWEEK](#), sin parámetros o la macro equivalente [MonthCal\\_GetFirstDayOfWeek](#).

## Navegacion

Los controles de calendario disponen de dos pequeños botones con iconos en forma triangular que apuntan a la izquierda y derecha, y que permiten navegar a través de los meses hacia atrás y hacia delante. Por defecto, cada vez que se pulsa uno de esos botones el control retrocede o avanza un mes (o dependiendo de la vista, un año, una década o un siglo).

Podemos modificar ese comportamiento usando el mensaje [MCM\\_SETMONTHDELTA](#) que modificará el valor del salto. Por ejemplo, si nuestro control muestra tres meses, podemos hacer que con cada desplazamiento se muestren tres meses anteriores o posteriores asignando el valor 3 al delta.

```
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_SETMONTHDELTA, (WPARAM)2, 0);
```

Para obtener el valor actual de delta se usa el mensaje [MCM\\_GETMONTHDELTA](#).

Las macros [MonthCal\\_SetMonthDelta](#) y [MonthCal\\_GetMonthDelta](#) son equivalentes, respectivamente.

## Hoy

Si no se ha especificado el estilo MCS\_NOTODAY, el control mostrará una línea que si es pulsada por el usuario seleccionará la fecha actual, actualizando el control para que esa fecha sea visible, si es necesario.

Por defecto, la fecha para 'hoy' es la fecha local actual, pero podemos modificar esa fecha usando el mensaje [MCM\\_SETTODAY](#), indicando en lParam el nuevo valor para la fecha de 'hoy' en una estructura [SYSTEMTIME](#).

```
SYSTEMTIME hoy;
...
/* Hasta nueva orden, hoy siempre será 15 de marzo de 2022 */
hoy.wDay = 15;
hoy.wMonth = 3;
hoy.wYear = 2022;
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_SETTODAY, 0, (LPARAM)hoy);
```

Si modificamos esta fecha el control no la actualizará automáticamente cuando pase la media noche. Para volver a usar el valor por defecto se debe usar el mensaje con un valor cero para lParam.

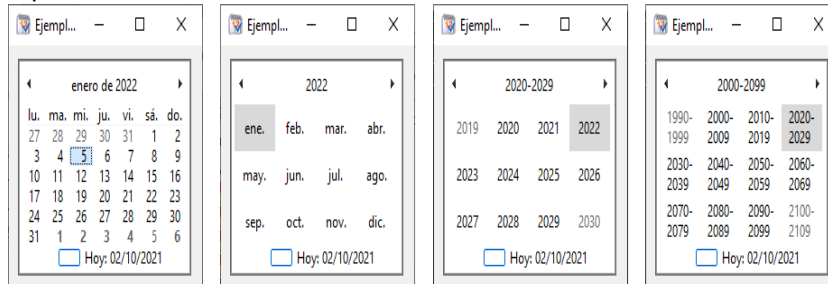
Usar la macro [MonthCal\\_SetToday](#) es equivalente.

Para recuperar el valor actual de 'hoy' se puede usar el mensaje [MCM\\_GETTODAY](#), indicando en lParam una estructura [SYSTEMTIME](#) válida que recibirá el valor de la fecha.

También se puede usar la macro equivalente [MonthCal\\_GetToday](#).

## Vistas

Además de la vista por defecto, que muestra los días de cada mes y que es la que permite seleccionar una fecha concreta, existen otras vistas que nos permiten hacer una navegación más rápida.



Vistas: mensual, anual, de década, de siglo.

Durante la ejecución del programa podemos cambiar de vista pulsando sobre la primera línea del calendario, que contiene el texto de 'mes' de 'año', eso cambia la vista a la anual, y la leyenda 'año', mostrando los nombres de los doce meses.

Pulsando en un mes volveremos a la vista mensual, pulsando en la leyenda del 'año' pasaremos a la vista de década, que mostrará doce años, aunque el primero y el último estarán en gris, y la leyenda mostrará 'año inicio'-'año final' de la década.

De nuevo, pulsando sobre un año volveremos a la vista anual, y pulsando sobre la leyenda pasaremos a la vista de siglo. Se mostrarán doce rangos de diez años, estando el primero y último en gris, y la leyenda mostrará 'año inicio'-'año final' del siglo. Al pulsar sobre una década volveremos a la vista de décadas.

La vista también se puede modificar desde la aplicación usando el mensaje [MCM\\_SETCURRENTVIEW](#), indicando en IParam una de las constantes de vista:

- Vista mensual: MCMV\_MONTH.
- Vista anual: MCMV\_YEAR.
- Vista de década: MCMV\_DECADE.
- Vista de siglo: MCMV\_CENTURY.

```
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_SETCURRENTVIEW, 0, (LPARAM)MCMV_CENTURY);
```

También se puede usar la macro equivalente [MonthCal\\_SetCurrentView](#).

Para obtener el tipo de vista actual de un control se puede usar el mensaje [MCM\\_GETCURRENTVIEW](#), o la macro correspondiente [MonthCal\\_GetCurrentView](#).

## Formato de juego de caracteres

Se puede cambiar la bandera de formato Unicode usando el mensaje [MCM\\_SETUNICODEFORMAT](#) indicando en wParam el valor de la bandera, cero para caracteres ANSI y distinto de cero para caracteres Unicode.

La macro [MonthCal\\_SetUnicodeFormat](#) también sirve para lo mismo.

El mensaje [MCM\\_GETUNICODEFORMAT](#) obtiene el valor de la bandera de formato Unicode. Al igual que la macro [MonthCal\\_GetUnicodeFormat](#).

## Puntos de prueba

Un punto de prueba, o "test point" es una función que nos permite averiguar a qué zona en concreto pertenece un punto determinado de la pantalla. El mensaje `MCM_HITTEST` nos permite probar puntos, pasando en `lParam` una estructura `MCHITTESTINFO`.

La estructura debe ser inicializada antes de enviar el mensaje. El miembro `cbSize` debe contener el tamaño de la estructura.

```
El miembro pt, de tipo POINT contendrá las coordenadas del punto a probar.

MCHITTESTINFO mcTI;
...
mcTI.cbSize = sizeof(MCHITTESTINFO);
mcTI.pt.x = 14;
mcTI.pt.y = 43;
SendDlgItemMessage(hwnd, ID_CALENDAR1, MCM_HITTEST, 0, (LPARAM)&mcTI);
```

El resto de los miembros son de salida:

- *uHit*: será una constante que indica la zona concreta en la que está el punto. (Ver `MCHITTESTINFO` para una lista de los posibles valores).
- *st*: es una estructura `SYSTEMTIME` que devolverá la fecha correspondiente a la zona donde está el punto, si a esa zona le corresponde una fecha.
- *rc*: es una estructura `RECT` que devuelve el área de la zona a la que pertenece el punto.
- *iOffset*: cuando hay más de un calendario en el control, indica el desplazamiento de aquel al que pertenece el punto.
- *iRow* e *iCol*: fila y columna concreta en la que está el punto, si está en una de las casillas.

Todos los desplazamientos, *iOffset*, *iRow* e *iCol* empiezan a contar desde cero.

La macro `MonthCal_HitTest` es equivalente.

## Notificaciones

Veremos ahora otros códigos de notificación que puede enviar el control de calendario a través de un mensaje `MM_NOTIFY`.

### Cambio de selección

Cada vez que el usuario cambie la selección de la fecha o la selección cambie automáticamente al navegar por el control o como consecuencia de un mensaje `MCM_SETCURSEL` o `MCM_SETSELRANGE`, se generará un código de notificación `MCN_SELCHANGE`.

En `lParam` recibiremos un puntero a una estructura `NMSELCHANGE`, en el miembro *nmhdr*, que es una estructura `NMHDR` con información sobre la notificación: manipulador de ventana del control, su identificador y el código de notificación.

También recibiremos dos miembros *stSelStart* y *stSelEnd* de tipo `SYSTEMTIME` con las fechas de inicio y final seleccionadas.

```
/* No se pueden seleccionar sábados o domingos */
NMHDR* pnmhdr;
NMSELCHANGE* pnmSC;
...
case WM_NOTIFY:
    pnmhdr = (NMHDR*)lParam;
    switch(pnmhdr->code) {
```

```

        case MCN_SELCHANGE:
            pnmSC = (NMSELCHANGE*)lParam;
            if (pnmSC->stSelStart.wDayOfWeek==6) {
                pnmSC->stSelStart.wDay--;
            }
            if (pnmSC->stSelStart.wDayOfWeek==0) {
                pnmSC->stSelStart.wDay++;
            }
            SendDlgItemMessage (hwnd, ID_CALENDAR1, MCM_SETCURSEL, 0, (LPARAM)&pnmSC->stSelStart);
            break;
        ...
    }
    return 0;

```

## Selección

La notificación [MCN\\_SELECT](#) es similar a [MCN\\_SELCHANGE](#), con la diferencia de que sólo se envía si se trata de una selección explícita del usuario. Es decir, cuando el usuario hace doble click sobre una fecha concreta, o cuando pulsa sobre la zona 'hoy' para seleccionar la fecha actual.

En lParam recibiremos un puntero a una estructura [NMSELCHANGE](#), igual que con el mensaje de notificación [MCN\\_SELCHANGE](#).

## Cambio de vista

Cuando se produzca un cambio de vista se enviará un código de notificación [MCN\\_VIEWCHANGE](#), y en lParam un puntero a una estructura [NMVIEWCHANGE](#).

La estructura contiene un miembro *nmhdr*, que es una estructura [NMHDR](#) con información sobre el código de notificación y el control que lo envía.

Además contiene dos miembros de tipo DWORD, *dwOldView* y *dwNewView* que contendrán los valores anterior y posterior del tipo de vista del control.

```

/* No se puede cambiar de vista */
NMHDR* pnmhdr;
NMSELCHANGE* pnmSC;
...
case WM_NOTIFY:
    pnmhdr = (NMHDR*)lParam;
    switch (pnmhdr->code) {
        case MCN_VIEWCHANGE:
            pnmVC = (NMVIEWCHANGE*)lParam;
            SendDlgItemMessage (hwnd, ID_CALENDAR1, MCM_SETCURRENTVIEW, 0,
(LPARAM) MCMV_MONTH);
            break;
    }

```

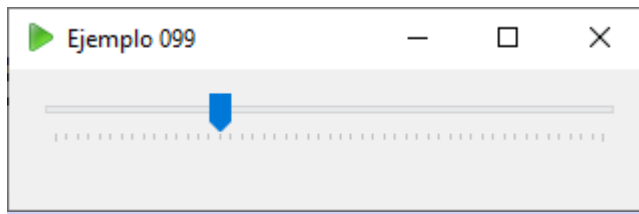
## Operaciones de arrastre

Si la aplicación implementa operaciones de drag and drop al recibir un mensaje de notificación [NM\\_RELEASEDCAPTURE](#) debe iniciar una de ellas.

## Ejemplo 98

# Capítulo 57 Control Trackbar

podemos considerar el control de trackbar como una variante del control scrollbar. Su funcionamiento es muy parecido, aunque en general se orientan a otro tipo de entradas de datos por parte del usuario, como asignar posiciones; también sirven para indicar el estado de progreso de una tarea o proceso.



El ejemplo más claro es en la reproducción de archivos multimedia, ya sean de sonido o de video. Este tipo de controles nos indica el punto en que se encuentra la reproducción, pero también permite elegir el punto en que queremos continuar la reproducción, avanzar o retroceder ciertos valores, etc.

Como en todos los controles comunes que estamos viendo, hay que asegurarse de que la DLL ha sido cargada invocando a la función [InitCommonControlsEx](#) indicando el valor de bandera `ICC_BAR_CLASSES` en el miembro `dwICC` de la estructura [INITCOMMONCONTROLSEX](#) que pasaremos como parámetro.

```
INITCOMMONCONTROLSEX icCE;  
...  
icCE.dwSize = sizeof(INITCOMMONCONTROLSEX);  
icCE.dwICC = ICC_BAR_CLASSES;  
InitCommonControlsEx(&icCE);
```

## Insertar durante la ejecución

Esto ya es rutina. Como todos los controles, los trackbar también se pueden insertar en una ventana o diálogo durante la ejecución. Tan sólo tendremos que usar las funciones [CreateWindow](#) o



[CreateWindowEx](#) e indicar en la clase de ventana el valor `TRACKBAR_CLASS`:

```
hFont = CreateFont(-14, 0, 0, 0, 0, FALSE, FALSE, FALSE,
1, 0, 0, 0, 0, ("Ms Shell Dlg"));
CreateWindowEx(0, TRACKBAR_CLASS, NULL,
WS_CHILD | WS_VISIBLE | WS_TABSTOP | TBS_HORZ,
10, 10, 300, 50,
hwnd, (HMENU)ID_TRACKBAR,
hInstance, NULL);
SendDlgItemMessage(hwnd, ID_TRACKBAR, WM_SETFONT,
(WPARAM)hFont, MAKELPARAM(FALSE, 0));
```

Hay varios estilos específicos para este tipo de control, pero los más básicos son los que afectan a la orientación: horizontal (`TBS_HORZ`) o vertical (`TBS_VERT`), aunque si no se especifica ninguno, el valor por defecto es horizontal.

En el parámetro *hMenu*, como siempre, indicaremos el identificador del control.

En principio, no sería necesario modificar la fuente, ya que este control no muestra ningún texto. (Ya veremos si esto es así). En cualquier caso, si se usa una fuente hay que recordar liberar el recurso antes de terminar el programa, usando [DeleteObject](#).

## Insertar desde fichero de recursos

Se usa un control general [CONTROL](#), con la clase `TRACKBAR_CLASS`, y los estilos generales y específicos que queramos aplicar.

```
LANGUAGE LANG_NEUTRAL, SUBLANG_NEUTRAL
IDD_DIALOG1 DIALOG 0, 0, 341, 159
STYLE DS_3DLOOK | DS_CENTER | DS_MODALFRAME | DS_SHELLFONT |
WS_CAPTION | WS_VISIBLE | WS_POPUP | WS_SYSMENU
CAPTION "Dialog"
FONT 8, "Ms Shell Dlg"
```

```

{
    CONTROL        "", 0, TRACKBAR_CLASS, WS_TABSTOP |
    TBS_AUTOTICKS | TBS_BOTH | TBS_TOOLTIPS, 21, 20, 304, 15,
    WS_EX_LEFT
    PUSHBUTTON     "Cancel", IDCANCEL, 285, 138, 50, 14, 0,
    WS_EX_LEFT
    DEFPUSHBUTTON  "OK", IDOK, 285, 121, 50, 14, 0,
    WS_EX_LEFT
}

```

## Partes del control trackbar



Partes de un trackbar

El deslizador es el cursor que desplaza el usuario para elegir un valor. El canal es la ranura sobre la que se desplaza el deslizador o thumb. Las marcas (o tics) son una escala de ayuda o guía para seleccionar valores.

## Establecer rango de valores

Si no se especifica, el rango de valores por defecto es de 0 a 100. Es decir, el valor mínimo del deslizador corresponde al valor 0, y el máxima al valor 100.

Este rango se puede modificar mediante el mensaje [TBM\\_SETRANGE](#). En wParam indicaremos si el control debe redibujarse, y en lParam, en la palabra de menor peso el límite inferior y en la palabra de mayor peso el límite superior.

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETRANGE,  
(WPARAM) FALSE, MAKELPARAM(100, 3000));
```

Al usar un DWORD para establecer ambos límites, el rango máximo que se puede establecer con este mensaje es de 0 a 35535, es decir, 16 bits.

Si necesitamos establecer valores para cualquiera de los extremos del margen mayores del ámbito de un WORD, disponemos de la pareja de mensaje [TBM\\_SETRANGEMIN](#) y [TBM\\_SETRANGEMAX](#). De nuevo, en wParam indicaremos si queremos redibujar el control, y en lParam el límite inferior o superior del rango, respectivamente.

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETRANGEMIN,  
(WPARAM) FALSE, (LPARAM) 45000);  
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETRANGEMAX,  
(WPARAM) FALSE, (LPARAM) 50000);
```

Para obtener los valores de los límites del rango actuales de un control trackbar disponemos de dos mensajes. [TBM\\_GETRANGEMIN](#) para obtener el límite inferior, y [TBM\\_GETRANGEMAX](#) para obtener el límite superior.

## Modificar y leer posición del deslizador

Para asignar una posición al deslizador disponemos del mensaje [TBM\\_SETPOS](#) indicando en wParam si queremos actualizar el control en pantalla, y en lParam la nueva posición del cursor.

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETPOS,  
(WPARAM) TRUE, (LPARAM) posicion);
```

Hay otro mensaje para modificar la posición del deslizador, [TBM\\_SETPOSNOTIFY](#), aunque en este caso no se usa el parámetro wParam, y se enviará un código de notificación [TRBN\\_THUMBPOSCHANGING](#).

Para obtener la posición actual del deslizador usaremos el mensaje [TBM\\_GETPOS](#).

```
DWORD pos;  
...  
pos = SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_GETPOS,  
0, 0);
```

## Estilos

Podemos agrupar los [estilos](#) propios de los controles trackbar en varias categorías:

### Que afectan a la orientación

Podemos elegir la orientación del control, horizontalmente con el estilo TBS\_HORZ, o verticalmente con el estilo TBS\_VERT.

Además de afectar al aspecto visual del control, también influye en el tipo de mensajes que recibiremos cuando el usuario interactúe con él. Los trackbars horizontales recibirán mensaje [WM\\_HSCROLL](#) y los verticales [WM\\_VSCROLL](#), cuando usemos las teclas del cursor o hagamos click con el ratón en el canal del control.

### Relacionados con las marcas

Con el estilo TBS\_AUTOTICKS se creará una marca automáticamente para cada incremento de valor.

Por el contrario, el estilo TBS\_NOTICKS impide que se muestren marcas, incluyendo la primera y la última.

Para controles trackbar horizontales podemos elegir mostrar las marcas encima, con TBS\_TOP o debajo, con TBS\_BOTTOM.

De forma análogo, para controles trackbar verticales, podemos elegir mostrar las marcas a la izquierda, con el estilo TBS\_LEFT, o a la derecha, con el estilo TBS\_RIGHT.

Si queremos mostrar las marcas a ambos lados, independientemente de la orientación, disponemos del estilo TBS\_BOTH.

Estos estilos afectan también al aspecto visual del deslizador. Cuando sólo se muestren marcas a uno de los lados, el extremo correspondiente del deslizador acabará en punta. Cuando se muestren a ambos lados ninguno de los extremos tendrá punta.

## Colocar marcas

El mensaje **TBM\_SETTIC** nos permite añadir marcas individuales en las posiciones indicadas en lParam. Por ejemplo, el siguiente código introduce dos marcas, una para el valor 25 y otra en el 50:

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTIC, 0,
(LPARAM) 25);
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTIC, 0,
(LPARAM) 50);
```

El mensaje **TBM\_SETTICFREQ** permite situar marcas cada cierto número de valores, indicando el periodo en el parámetro wParam. Por ejemplo, si nuestro trackbar tiene valores entre 0 y 100, e indicamos una frecuencia de 5, se situará una marca cada 5 valores, es decir, 20 marcas equidistantes:

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTICFREQ,
(WPARAM) 5, 0);
```

## Obtener marcas

El mensaje **TBM\_GETNUMTICS** nos devuelve el número de marcas actualmente en el control trackbar. Esta cuenta incluye las marcas de inicio y final que se insertan por defecto.

Con el mensaje **TBM\_GETTIC** podemos obtener la posición lógica asociada a una marca, indicando en wParam su valor de índice. La posición lógica es el valor asociado a la marca. Este mensaje parece funcionar de forma diferente para marcas insertadas individualmente con **TBM\_SETTIC** y con marcas insertadas automáticamente con **TBM\_SETTICFREQ**.

En el primer caso se ignorarán las marcas de inicio y final, por lo tanto los valores válidos son dos menos que el valor obtenido por el mensaje **TBM\_GETNUMTICS**. Por ejemplo, si hemos insertado tres marcas en las posiciones 25, 50 y 75, el valor obtenido por **TBM\_GETNUMTICS** será 5, pero sólo podremos usar con este mensaje los valores 0 a 2. Obtendremos los valores 25, 50 y 75 al usar el mensaje **TBM\_GETTIC**.

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTIC, 0,
(LPARAM)25);
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTIC, 0,
(LPARAM)50);
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTIC, 0,
(LPARAM)75);
printf("%d\n", SendDlgItemMessage(hwnd, ID_TRACKBAR,
TBM_GETNUMTICS, 0, 0));
for(i=0; i < SendDlgItemMessage(hwnd, ID_TRACKBAR,
TBM_GETNUMTICS, 0, 0); i++)
printf("%d - %d\n", i, SendDlgItemMessage(hwnd,
ID_TRACKBAR, TBM_GETTIC, (LPARAM)i, 0));
```

Salida:

```
5
0 - 25
```

```
1 - 50
2 - 75
3 - -1
4 - -1
```

En el caso de usar **TBM\_SETTICFREQ** sí se tendrán en cuenta las marcas de inicio y final, pero los valores obtenidos no serán los valores lógicos esperados:

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTICFREQ,
(WPARAM)25, 0);
printf("%d\n", SendDlgItemMessage(hwnd, ID_TRACKBAR,
TBM_GETNUMTICS, 0, 0));
for(i=0; i < SendDlgItemMessage(hwnd, ID_TRACKBAR,
TBM_GETNUMTICS, 0, 0); i++)
printf("%d - %d\n", i, SendDlgItemMessage(hwnd,
ID_TRACKBAR, TBM_GETTIC, (WPARAM)i, 0));
```

La salida ahora será:

```
5
0 - 1
1 - 2
2 - 3
3 - 4
4 - 5
```

Ignoro si esto es intencionado, pero resulta raro.

El mensaje **TBM\_GETPTICS** nos da exactamente la misma información, pero en lugar de tener que indicar el índice de la marca a recuperar, nos devuelve la dirección de un array con los valores de todas las marcas. Este array será válido mientras no se añadan o eliminen marcas.

```
DWORD* pos;
...
```

```

    SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTICFREQ,
(WPARAM)25, 0);
    printf("%d\n", SendDlgItemMessage(hwnd, ID_TRACKBAR,
TBM_GETNUMTICS, 0, 0));
    pos = SendDlgItemMessage(hwnd, ID_TRACKBAR,
TBM_GETPTICS, 0, 0);
    for(i=0; i < SendDlgItemMessage(hwnd, ID_TRACKBAR,
TBM_GETNUMTICS, 0, 0); i++)
        printf("%d - %d\n", i, pos[i]);

```

Dará la salida:

```

5
0 - 1
1 - 2
2 - 3
3 - 4
4 - 5

```

## Eliminar marcas

Mediante el mensaje **TBM\_CLEAR** se eliminarán todas las marcas insertadas, salvo las de inicio y final. En wParam indicaremos si queremos que se actualice el control en pantalla.

## Posiciones de marcas

El mensaje **TBM\_GETTICPOS** obtiene la distancia en coordenadas de cliente desde el extremo izquierdo del área de cliente del control.

De nuevo, este mensaje funciona de forma diferente con marcas insertadas manualmente y con las automáticas.

Si las marcas fueron insertadas manualmente, obtendremos desplazamientos en coordenadas de cliente:

```

    SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTIC, 0,
(LPARAM)25);

```



```

    SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTIC, 0,
(LPARAM)50);
    SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTIC, 0,
(LPARAM)75);
    for(i=0; i < SendDlgItemMessage(hwnd, ID_TRACKBAR,
TBM_GETNUMTICS, 0, 0); i++)
        printf("%d\n", SendDlgItemMessage(hwnd, ID_TRACKBAR,
TBM_GETTICPOS, (WPARAM)i, 0));

```

La salida tiene la forma:

```

80
150
219
-1
-1

```

Por el contrario, con marcas insertadas automáticamente dan una salida de otro tipo:

```

    SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTICFREQ,
(WPARAM)25, 0);
    for(i=0; i < SendDlgItemMessage(hwnd, ID_TRACKBAR,
TBM_GETNUMTICS, 0, 0); i++)
        printf("%d\n", SendDlgItemMessage(hwnd, ID_TRACKBAR,
TBM_GETTICPOS, (WPARAM)i, 0));

```

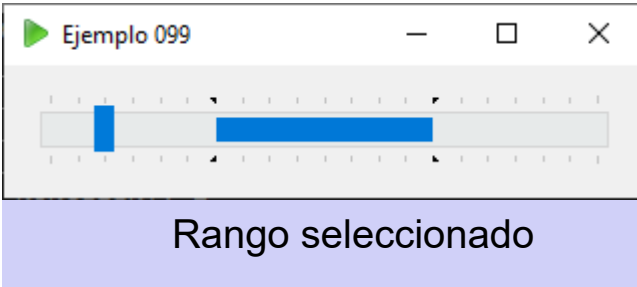
Da la salida:

```

14
17
19
22
25
28

```

## Relacionados con los rangos



Los controles trackbar también disponen de la opción de seleccionar un rango de valores seleccionados que se mostrarán resaltados, dentro del rango de valores

totales disponibles. Para poder seleccionar un rango de valores, el control debe tener el estilo `TBS_ENABLESELRANGE`. Además, las marcas en las posiciones de principio y final de un rango de seleccionado son mostrados como triángulos, en lugar de líneas verticales.

Para seleccionar un rango disponemos de varios mensajes. El más sencillo es `TBM_SETSEL`. En `wParam` indicaremos si queremos que el control se actualice en pantalla para reflejar el rango seleccionado, en la palabra de menor peso de `lParam` indicaremos el límite inferior y en la de mayor peso el límite superior.

Este mensaje será suficiente en la mayor parte de las situaciones, pero dado que usaremos un valor de 32 bits para indicar los dos extremos, no nos servirá si los valores que queremos resaltar son mayores de un valor de 16 bits.

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETSEL,  
(WPARAM) TRUE, MAKELPARAM(30, 70));
```

Para solventar esta limitación disponemos de una pareja de mensajes `TBM_SETSELSTART` y `TBM_SETSELEND`, el primero para establecer el límite inferior y el segundo para el superior. En `wParam` indicaremos si queremos que el control se actualice en pantalla, y en el segundo un valor de 32 bits para el límite indicado.

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETSELSTART,  
(WPARAM) TRUE, (LPARAM) 30);
```

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETSELEND,  
(WPARAM) TRUE, (LPARAM) 70);
```

Además, el mensaje [TBM\\_CLEARSEL](#) elimina la selección. En wParam seguiremos indicando si queremos o no actualizar el control.

Para obtener los límites de la selección actual disponemos de la pareja de mensajes [TBM\\_GETSELSTART](#) y [TBM\\_GETSELEND](#) que nos devuelven, respectivamente, el límite inferior y el superior.

## Relacionados con el deslizador

Si se usa el estilo `TBS_FIXEDLENGTH`, podremos modificar el tamaño de deslizador del control trackbar.

Para ello disponemos del mensaje [TBM\\_SETTHUMBLENGTH](#), al que pasaremos la longitud del deslizador deseada, en pixels, en el parámetro wParam.

Para obtener la longitud actual del deslizador se usa el mensaje [TBM\\_GETTHUMBLENGTH](#).

El estilo `TBS_NOTHUMB` oculta el deslizador.

## Tooltips

Si se usa el estilo `TBS_TOOLTIPS`, el control trackbar soportará tooltips. Por defecto, se crea un control tooltip que muestra la posición actual del deslizador.

Con el mensaje [TBM\\_SETTIPSIDE](#) podemos elegir la posición en la que se mostrará el tooltip, indicándolo en wParam. Para los horizontales podemos elegir encima o debajo, `TBTS_TOP` o `TBTS_BOTTOM`, y para los verticales izquierda o derecha, `TBTS_LEFT` o `TBTS_RIGHT`.

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETTIPSIDE,  
(WPARAM) TBTS_BOTTOM, 0);
```

El mensaje **TBM\_SETTOOLTIPS** se puede usar para asignar un control tooltip previamente creado al control trackbar. Para ello pasaremos en wParam un manipulador de ventana del control tooltip.

```
hToolNew = CreateWindowEx(WS_EX_TOOLWINDOW,
    TOOLTIPS_CLASS, NULL,
    WS_POPUP | TTS_ALWAYSTIP,
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT,
    hwnd, NULL, hInstance, NULL);
if(hToolNew) {
    SetWindowPos(hToolNew, HWND_TOPMOST, 0, 0, 0, 0,
    SWP_NOMOVE | SWP_NOSIZE | SWP_NOACTIVATE);
    SendMessage(hToolNew, TTM_SETMAXTIPWIDTH, 0, 150);

    toolInfo.cbSize = sizeof(toolInfo);
    toolInfo.hwnd = hCtrl;;
    toolInfo.hinst = hInstance;
    toolInfo.uFlags = TTF_IDISHWND | TTF_SUBCLASS;
    toolInfo.uId = (UINT_PTR)GetDlgItem(hwnd,
ID_TRACKBAR);
    toolInfo.lpszText = LPSTR_TEXTCALLBACK;
    SendMessage(hToolNew, TTM_ADDTOOL, 0,
    (LPARAM)&toolInfo);

    hToolPrev = SendDlgItemMessage(hwnd, ID_TRACKBAR,
    TBM_GETTOOLTIPS, 0, 0);
    SendDlgItemMessage(hwnd, ID_TRACKBAR,
    TBM_SETTOOLTIPS, (WPARAM)hToolNew, 0);
}
```

Por otra parte, tendremos que procesar la notificación **TTN\_GETDISPINFO** para asignar el texto al tooltip cuando lo requiera:

```
case WM_NOTIFY:
    pnmhdr = (LPNMHDR)lParam;
    switch(pnmhdr->code) {
```

```

        case TTN_GETDISPINFO:
            pnmtdispinfo = (LPNMTTDISPINFO)pnmhdr;
            if(GetDlgCtrlID((HWND)pnmtdispinfo->hdr.idFrom)
== ID_TRACKBAR) {
                sprintf(cad, "Valor: %d", SendMessage(hCtrl,
TBM_GETPOS, 0, 0));
                strcpy(pnmtdispinfo->lpszText, cad);
            }
            break;
        }
    }
    break

```

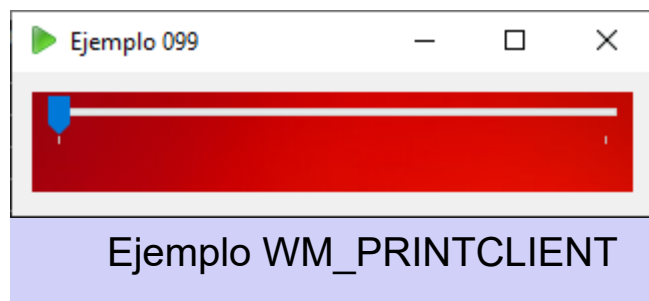
Estas notificaciones también funcionarán con los tooltips creados por defecto.

Si fuera necesario, podemos recuperar el manipulador de ventana del tooltip actualmente asociado a un control trackbar mediante el mensaje [TBM\\_GETTOOLTIPS](#).

## Otros estilos

Hay otros estilos menos útiles, por ejemplo TBS\_REVERSED se usa para trackbars "invertidas", donde un número más pequeño indica "mayor" y un número más grande indica "menor." Esto no afecta al control para nada, es sólo una etiqueta que puede ser consultada para determinar si un trackbar es normal o inverso.

Si no se indica el estilo TBS\_DOWNISLEFT, en un control trackbar la tecla de dirección 'abajo' equivale a 'derecha' y 'arriba' a 'izquierda'. Utilizar este estilo invierte el comportamiento por defecto de modo que 'abajo' equivale a 'izquierda' y 'arriba' a 'derecha'.



Con el estilo TBS\_TRANSPARENTBKGD el responsable de pintar el fondo es la ventana a través de un mensaje [WM\\_PRINTCLIENT](#).

```

static HBITMAP hBitmapRes;
static HWND hCtrl;
POINT punto;
RECT wre;
HDC memDC;
...
    case WM_CREATE:
        hCtrl = CreateWindowEx(0, TRACKBAR_CLASS, NULL,
            WS_CHILD | WS_VISIBLE | WS_TABSTOP |
TBS_TOOLTIPS | TBS_TRANSPARENTBKGND,
            10,10,300,50,
            hwnd, (HMENU)ID_TRACKBAR,
            hInstance, NULL);
        hBitmapRes = LoadBitmap(hInstance, "rojo");
...
    case WM_PRINTCLIENT:
        // Obtener posición del control en coordenadas
de pantalla:
        GetWindowRect(hCtrl, &wre);
        // Calcular tamaño:
        wre.right -= wre.left;
        wre.bottom -= wre.top;
        // Convertir posición a coordenadas de cliente:
        punto.x = wre.left;
        punto.y = wre.top;
        ScreenToClient(hwnd, &punto);
        // Mostrar mapa de bits:
        hdc = (HDC)wParam;
        memDC = CreateCompatibleDC(hdc);
        SelectObject(memDC, hBitmapRes);
        BitBlt(hdc, punto.x, punto.y, wre.right,
wre.bottom, memDC, 0, 0, SRCCOPY);
        DeleteDC(memDC);
        break;
...
    case WM_DESTROY:
        DeleteObject(hBitmapRes);

```

**Nota:**

El mensaje **WM\_PRINTCLIENT** sólo se envía si están activos los estilos visuales, usando el manifiesto adecuado.

## Desplazamientos

Al usar las teclas del cursor para desplazar el deslizador, por defecto, los incrementos y decrementos son de una unidad. Con las teclas de avance y retroceso de página los incrementos y decrementos son de 20 unidades por defecto. Los avances y retrocesos de página también se hacen haciendo click izquierdo sobre el canal.

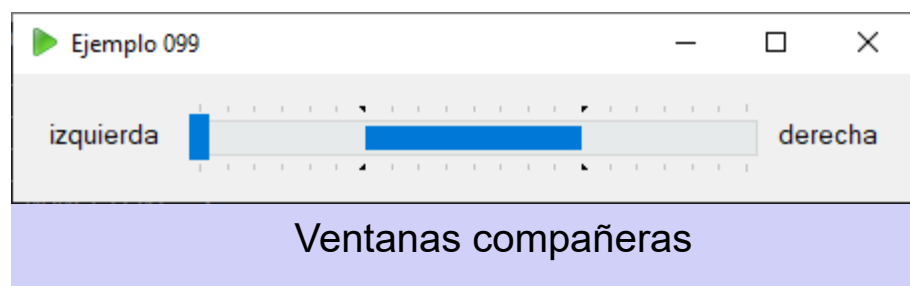
Estos valores se pueden modificar con el mensaje [TBM\\_GETLINESIZE](#), para cambiar el tamaño de línea y con [TBM\\_GETPAGESIZE](#) para cambiar el tamaño de página. En los dos mensajes se especifica el nuevo valor en el parámetro lParam.

```
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETLINESIZE,  
0, (LPARAM)2);  
SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETPAGESIZE,  
0, (LPARAM)10);
```

Para obtener los valores actuales de línea y página se usan los mensajes [TBM\\_SETLINESIZE](#) y [TBM\\_SETPAGESIZE](#).

## Ventanas compañeras

Se pueden añadir dos ventanas compañeras al control trackbar.



Generalmente serán controles estáticos, pero en realidad puede ser cualquier tipo de ventana.

Para insertar una ventana compañera se usa el mensaje [TBM\\_SETBUDDY](#). En wParam indicaremos un valor TRUE para insertar la ventana a la izquierda, si es un trackbar horizontal, o encima, si se trata de un trackbar vertical, o un valor FALSE para insertarla a la derecha o debajo, respectivamente.

En este ejemplo insertamos un botón como ventana compañera a la izquierda, y un texto estático a la derecha. Como siempre que insertamos controles en ejecución, si queremos modificar las fuentes deberemos enviar un mensaje [WM\\_SETFONT](#).

```
    HWND hctrl;
    HFONT hFont;
    ...

    hFont = CreateFont(-14, 0, 0, 0, 0, FALSE, FALSE, FALSE,
1, 0, 0, 0, 0, ("Ms Shell Dlg"));
    CreateWindowEx(0, TRACKBAR_CLASS, NULL, WS_CHILD |
WS_VISIBLE | WS_TABSTOP | TBS_TOOLTIPS,
110,10,300,50, hwnd, (HMENU)ID_TRACKBAR,
hInstance, NULL);
    SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETLINE SIZE,
0, (LPARAM)2);
    SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETPAGESIZE,
0, (LPARAM)10);
    hctrl = CreateWindowEx(0, "BUTTON", (LPSTR)"Botón",
WS_CHILD | WS_VISIBLE, 10, 10, 60, 24, hwnd,
(HMENU)(ID_BOTON), hInstance,
NULL);
    SendMessage(hctrl, WM_SETFONT, (WPARAM)hFont,
MAKELPARAM(FALSE, 0));
    SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETBUDDY,
(WPARAM)TRUE, (LPARAM)hctrl);
    hctrl = CreateWindowEx(0, "STATIC", "derecha", WS_CHILD
| WS_VISIBLE | SS_SIMPLE, 0,0,60,20, hwnd, NULL, hInstance,
NULL);
    SendMessage(hctrl, WM_SETFONT, (WPARAM)hFont,
MAKELPARAM(FALSE, 0));
    SendDlgItemMessage(hwnd, ID_TRACKBAR, TBM_SETBUDDY,
(WPARAM)FALSE, (LPARAM)hctrl);
```



El mensaje [TBM\\_GETBUDDY](#) sirve para obtener el manipulador de ventana de la compañera indicada en wParam, TRUE para la izquierda o superior, o false para la derecha o inferior, según se trate de un control trackbar horizontal o vertical, respectivamente.

## Delimitadores de áreas

Disponemos de un par de mensajes para obtener las áreas delimitadoras de ciertas partes de los controles trackbar.

El mensaje [TBM\\_GETCHANNELRECT](#) recupera un puntero con el rectángulo delimitador del canal de control en lParam.

El mensaje [TBM\\_GETTHUMBRECT](#) recupera un puntero con el rectángulo delimitador del deslizador en lParam.

## Formato de caracteres

Se puede modificar durante la ejecución el juego de caracteres usado por un control trackbar mediante el mensaje [TBM\\_SETUNICODEFORMAT](#), indicando en wParam un valor distinto de cero para usar caractere Unicode, o cero para usar caracteres ANSI.

Para recuperar el juego de caracteres Unicode usado por un control trackbar se usa el mensaje [TBM\\_GETUNICODEFORMAT](#).

Ignoro en qué influye usar uno u otro juego de caracteres en un control que no muestra texto, pero las opciones existen.

## Notificaciones

Si el control tiene el estilo TBS\_NOTIFYBEFOREMOVE se enviará al procedimiento de ventana de la ventana padre un código de notificación [TRBN\\_THUMBPOSCHANGING](#) cada vez que el usuario modifique la posición de deslizador, y antes de que el control

se actualice. Esto permite a la aplicación evitar que los cambios de posición se consoliden según nos convenga.

En `IParam` recibiremos un puntero a una estructura `NMTRBTHUMBPOSCHANGING` con la información necesaria sobre la nueva posición del deslizador, en el miembro `dwPos` y el modo en que se ha modificado en el miembro `nReason`.

El miembro `nReason` puede ser uno de los siguientes valores, dependiendo de cómo se haya modificado la posición del deslizador:

- `TB_LINEUP`: Se ha pulsado la tecla de `o`.
- `TB_LINEDOWN`: Se ha pulsado la tecla de dirección `o`.
- `TB_PAGEUP`: Se ha pulsado la tecla de `RePág` o se ha hecho click en el canal a la derecha o debajo de la posición actual del deslizador.
- `TB_PAGEDOWN`: Se ha pulsado la tecla de `AvPág` o se ha hecho click en el canal a la izquierda o encima de la posición actual del deslizador.
- `TB_THUMBTRACK`: Se está moviendo el deslizador con el ratón.
- `TB_TOP`: Se ha pulsador la tecla `Inicio`.
- `TB_BOTTOM`: Se ha pulsador la tecla `Fin`.
- `TB_THUMBPOSITION`: Este código está relacionado con la captura del ratón.
- `TB_ENDTRACK`: Este código está relacionado con la captura del ratón.

No hay notificación si el cambio de posición se produce como consecuencia de acciones de la rueda del ratón.

```
LPNMHDR pnmhdr;  
NMTRBTHUMBPOSCHANGING* pnmtpc;  
char cad[100];  
char* szReason[] = {  
    "TB_LINEUP",  
    "TB_LINEDOWN",  
    "TB_PAGEUP",  
    "TB_PAGEDOWN",
```

```

        "TB_THUMBPOSITION",
        "TB_THUMBTRACK",
        "TB_TOP",
        "TB_BOTTOM",
        "TB_ENDTRACK"
    };

    ...
    case WM_CREATE:
        CreateWindowEx(0, "STATIC", "", WS_CHILD |
WS_VISIBLE | SS_SIMPLE, 10, 70, 220, 20,
        hwnd, (HMENU) (ET_TRACKBAR), hInstance, NULL);

    ...
    case WM_NOTIFY:
        pnmhdr = (LPNMHDR) lParam;
        switch(pnmhdr->code) {
            case TRBN_THUMBPOSCHANGING:
                pnmtpc = (NMTRBTHUMBPOSCHANGING*) pnmhdr;
                sprintf(cad, "%100s", " ");
                SetWindowText(GetDlgItem(hwnd, ET_TRACKBAR),
cad);

                sprintf(cad, "%-25s -> %8d\n",
szReason[pnmtpc->nReason], pnmtpc->dwPos);
                SetWindowText(GetDlgItem(hwnd, ET_TRACKBAR),
cad);

                break;

```

#### Nota:

El código de notificación [TRBN\\_THUMBPOSCHANGING](#) sólo se envía si están activos los estilos visuales, usando el manifiesto adecuado y si el control tiene el estilo [TBS\\_NOTIFYBEFOREMOVE](#).

## Trackbar *custom draw*

Si queremos personalizar el aspecto de nuestro control trackbar deberemos procesar el código de notificación [NM\\_CUSTOMDRAW](#). En lParam recibiremos un puntero a una estructura

**NMCUSTOMDRAW** con la información necesaria para mostrar el control según nuestras preferencias.

El miembro *dwItemSpec* de esa estructura contendrá uno de los valores de **Custom Draw Values**, indicando qué parte del control se ha de actualizar.

## **Ejemplo 99**

# Glosario

Agruparemos en ésta página las palabras y siglas que se usan a menudo cuando se programa en Windows:

## **API (Application Programming Interface).**

El Win32 API son bibliotecas de C, aunque nadie nos impide usar lo con un compilador de C++. Contiene todas las funciones necesarias para programar para Windows.

Incluye: el fichero windows.h, constantes, funciones, mensajes, secuencias de escape de impresora y estructuras de datos.

## **Callback (retrollamada).**

Las funciones callback son funciones creadas por el programador. Windows utiliza estas funciones para que el programador pueda personalizar la respuesta a ciertos eventos o funciones del API. Es decir, son funciones de usuario que serán llamadas por el sistema.

## **Manipulador (Handle)**

En general se trata de números enteros que facilitan la manipulación de objetos Windows en llamadas a funciones. Windows puede obtener o modificar los datos del objeto a través de su manipulador, y resulta mucho más útil trabajar con números enteros que con punteros o estructuras.

## **Owner-draw**

Entre los términos de difícil traducción habituales en el API, uno de los más frecuentes, (y que no hemos querido ni intentado traducir) es el de owner-draw.

Literalmente significa "dibujado por el dueño". Se trata de un estilo que se aplica a los controles de Windows: controles edit, list box, botones, etc, que inhibe el tratamiento automático del control, en lo que se refiere a su representación gráfica, y le deja esa responsabilidad a la ventana padre o propietaria del control.

Esto permite personalizar el aspecto gráfico del control, por una parte, y el comportamiento general, en muchos casos.

## **OWL y MFC**

También existen bibliotecas de clases para programar en Windows, las más conocidas son OWL (Object Windows Library) de Borland y MFC (Microsoft Foundation Class) de Microsoft. Aquí no las usaremos, pero para algunos elementos de uso frecuente probablemente diseñaremos nuestras propias clases, y probablemente construyamos una biblioteca con ellas.

## **GDI (Graphics Device Interface).**

Funciones para el acceso a los gráficos. Permiten hacer programas gráficos independientes de hardware. Nuestros programas Windows funcionarían independientemente de la tarjeta gráfica, monitor o impresora que usemos. Y tendrán acceso a fuentes de caracteres y funciones para dibujar líneas y formas, manejo de mapas de bits, etc.

## **SDK (Software Development Kit).**

Contiene el Win32 API y los programas necesarios para la depuración y para la creación de ficheros de ayuda.

## **MAPI (Messaging Application Programming Interface).**

biblioteca que permite usar el correo electrónico desde nuestras aplicaciones para enviar mensajes.

## **MDI (Multiple Document Interface).**

Aplicaciones que pueden manejar varios documentos simultáneamente, estos documentos pueden ser todos del mismo tipo, aunque no necesariamente.

## **SDI (Single Document Interface).**

Aplicaciones que sólo pueden manejar un documento.

## **GUI (Graphic User Interface)**

Interface gráfico para el usuario, son las aplicaciones normales en Windows, el otro tipo son las aplicaciones de consola, que emulan una pantalla de texto como las de MS-DOS.

## **OEM (original equipment manufacturer)**

Fabricación del equipo original. Indica parámetros que dependen del fabricante del hardware.

# Tabla de contenido

- **Introducción**
  - Requisitos previos
  - Independencia de la máquina
  - Recursos
  - Ventanas
  - Eventos
  - Proyectos
  - Convenciones
  - Controles
- **1 Componentes de una ventana**
  - El borde de la ventana
  - Barra de título
  - Caja de minimizar
  - Caja de maximizar
  - Caja de cerrar
  - Caja de control de menú
  - Menú
  - Barra de menú
  - Barra de desplazamiento horizontal
  - Barra de desplazamiento vertical
  - El área de cliente
- **2 Notación húngara**
  - Ejemplos
- **3 Estructura de un programa Windows GUI**
  - Ficheros de cabecera
  - Prototipos
  - Función de entrada, WinMain
    - Parámetros de entrada de WinMain
    - Función WinMain típica
    - Declaración
    - Inicialización
    - Bucle de mensajes
  - Definición de funciones



- 4 El procedimiento de ventana
  - Sintaxis
  - Prototipo de procedimiento de ventana
  - Implementación de procedimiento de ventana simple
  - Primer ejemplo de programa Windows GUI
- 5 Menús 1
  - Usando las funciones para inserción ítem a ítem
  - Uso básico de MessageBox
  - Respondiendo a los mensajes del menú
  - Ejemplo 2
  - Ficheros de recursos
  - Cómo usar los recursos de menú
  - Ejemplo 3
- 6 Diálogo básico
  - Ficheros de recursos
  - Procedimiento de diálogo
  - Sintaxis
  - Prototipo de procedimiento de diálogo
  - Implementación de procedimiento de diálogo para nuestro ejemplo
  - Pasar parámetros a un cuadro de diálogo
  - Ejemplo 4
- 7 Control básico Edit
  - Fichero de recursos
  - El procedimiento de diálogo y los controles edit
  - Variables a editar en los cuadros de diálogo
  - Iniciar controles edit
  - Devolver valores a la aplicación
  - Añadir la opción de cancelar
  - Ejemplo 5
  - Editar números
  - Fichero de recursos para editar enteros
  - Variables a editar en los cuadros de diálogo
  - Iniciar controles edit de enteros
  - Devolver valores a la aplicación
  - Ejemplo 6
- 8 Control básico ListBox

- Ficheros de recursos
  - Iniciar controles listbox
  - Devolver valores a la aplicación
  - Ejemplo 7
- 9 Control básico Button
  - Ficheros de recursos
  - Iniciar controles button
  - Tratamiento de acciones de los controles button
  - Ejemplo 8
- 10 Control básico Static
  - Ficheros de recursos
  - Iniciar controles static
  - Tratamiento de acciones de los controles static
  - Ejemplo 9
- 11 Control básico ComboBox
  - Ficheros de recursos
  - Iniciar controles ComboBox
  - Devolver valores a la aplicación
  - Ejemplo 10
- 12 Control básico Scrollbar
  - Ficheros de recursos
  - Iniciar controles Scrollbar
  - Iniciar controles scrollbar: estructura SCROLLINFO
  - Procesar los mensajes procedentes de controles Scrollbar
  - Procesar mensajes de scrollbar usando SCROLLINFO
  - Devolver valores a la aplicación
  - Ejemplo 11
  - Ejemplo 12
- 13 Control básico Groupbox
  - Ficheros de recursos
  - Iniciar controles GroupBox
  - Devolver valores a la aplicación
  - Ejemplo 13
- 14 Control básico Checkbox
  - Ficheros de recursos
  - Iniciar controles CheckBox
  - Procesar mensajes de los CheckBox

- Devolver valores a la aplicación
  - Ejemplo 14
- 15 Control básico RadioButton
  - Ficheros de recursos
  - Iniciar controles RadioButton
  - Procesar mensajes de los RadioButtons
  - Devolver valores a la aplicación
  - Ejemplo 15
- 16 El GDI
  - Objetos del GDI
- 17 Objetos básicos del GDI: El Contexto de dispositivo, DC
  - Actualizar el área de cliente de una ventana, el mensaje WM\_PAINT
  - Colores
- 18 Objetos básicos del GDI: La pluma (Pen)
  - Plumitas de Stock
  - Plumitas cosméticas y geométricas
  - Crear una pluma
  - Seleccionar una pluma
  - Destruir una pluma
  - Ejemplo 16
- 19 Funciones para el trazado de líneas
  - Trazado de arcos, función Arc
  - Curvas Bézier
  - Funciones Poly<tipo>
  - Función LineDDA y funciones callback LineDDAProc
  - Ejemplo 17
- 20 Objetos básicos del GDI: El pincel (Brush)
  - Pinceles lógicos
    - Pinceles sólidos
    - Pinceles de Stock
    - Pinceles de tramas (Hatch)
    - Pinceles de patrones
  - Crear un pincel
  - Seleccionar un pincel
  - Destruir un pincel
  - Ejemplo 18

- 21 Funciones para el trazado de figuras rellenas
  - Pintando trozos de elipses, funciones Chord y Pie
  - Modos de relleno de polígonos
  - Ejemplo 19
- 22 Objetos básicos del GDI: La paleta (Palette)
  - Capacidades de Color de los dispositivos
  - Definiciones de valores de color
  - Aproximaciones de colores y mezclas de pixels (dithering)
  - Mezclas de colores (ROP)
  - Paletas de colores
  - La paleta por defecto
  - Paleta lógica
  - Paleta de sistema
  - Ejemplo 20
- 23 Objetos básicos del GDI: El Mapa de Bits (Bitmap)
  - Tipos de mapas de bits
  - Crear un mapa de bits
  - Fichero de recursos
  - Fichero BMP
  - Mostrar un mapa de bits
  - Funciones de visualización de mapas de bits
    - BitBlt
    - StretchBlt
    - PlgBlt
    - MaskBlt
  - Códigos ROP ternarios
  - Códigos ROP cuádruples
  - Pinceles creados a partir de mapas de bits
    - PatBlt
    - ExtFloodFill
  - Estructuras de datos
    - BITMAP
  - Modos de estiramiento (stretch modes)
  - Mapas de bits de stock
  - Ejemplo 21
- 24 Objetos básicos del GDI: La Fuente (Font)
  - Mostrar un texto simple

- Cambiar el color del texto
- Ejemplo 22
- Crear fuentes personalizadas
  - Altura y anchura media de carácter
  - El ángulo de escape
  - El ángulo de orientación
  - Peso
  - Cursiva
  - Subrayado
  - Tachado
  - Conjunto de caracteres
  - Precisión de salida
  - Precisión de recorte
  - Calidad
  - Paso y familia
  - Nombre
- Fuentes de stock
- Alineamientos de texto
- Separación de caracteres
- Medidas de cadenas
- Justificar texto
- Ejemplo 23
- 25 Objetos básicos del GDI: Rectángulos y Regiones
  - Rectángulos
  - Funciones para trabajar con rectángulos
    - Asignar rectángulos
    - Comparaciones de rectángulos
    - Modificar rectángulos
    - Operaciones con rectángulos
  - Ejemplo 24
  - Regiones
  - Funciones para regiones
    - Crear regiones
    - Combinar regiones
    - Comparar regiones
    - Rellenar regiones
    - Mover una región

- Comprobar posiciones
  - Destruir regiones
- Ejemplo 25
- 26 Objetos básicos del GDI: El camino (Path)
  - Crear un camino
  - Operaciones con caminos
  - Ejemplo 26
- 27 Objetos básicos del GDI: El recorte (Clipping)
  - Regiones de recorte y el mensaje WM\_PAINT
  - Funciones relacionadas con el recorte
  - Seleccionar regiones de recorte
  - Caminos de recorte
  - Ejemplo 27
- 28 Objetos básicos del GDI: Espacios de coordenadas y transformaciones
  - Definiciones
  - Transformaciones
    - Traslaciones
    - Cambio de escala
    - Rotaciones
    - Cambio de ejes
    - Reflexiones
  - Aplicar transformaciones
  - Combinar transformaciones
  - Cambios de escala y plumas
  - Ejemplo 28
  - Ventanas y viewports
    - Extensiones
    - Orígenes
  - Mapeos
  - Modos de mapeo predefinidos
  - Modo por defecto
  - Transformaciones definidas por el usuario
  - Modos gráficos y sentido de los arcos
  - Otras funciones
  - Ejemplo 29
- 29 Objetos básicos del GDI: Plumillas geométricas

- Atributos de las plumas geométricas
  - Anchura
  - Estilo de línea
  - Color
  - Patrón
  - Rayado
  - Estilo de final (tapón)
  - Estilo de unión
- Crear una pluma geométrica
- Seleccionar una pluma geométrica
- Destruir una pluma geométrica
- 30 Objetos básicos de usuario: El Caret
  - Recibir y perder el foco
  - Crear y destruir carets
  - Mostrar y ocultar carets
  - Procesar mensajes WM\_PAINT
  - Cambiar posición de un caret
  - Cambiar velocidad de parpadeo de un caret
  - Ejemplo 31
- 31 Objetos básicos del usuario: El icono
  - Punto activo
  - Tamaños
  - Asociar iconos a una aplicación
  - Tipos
  - Iconos en ficheros de recursos
  - Iconos en controles estáticos
  - Mostrar iconos
  - Destrucción de iconos
  - Ejemplo 32
- 32 Objetos básicos del usuario: El cursor
  - Cursor de clase
  - Cursores de recursos
  - Cursores estándar
  - Similitud entre iconos y cursores
  - El punto activo (Hot Spot)
  - Crear cursores
  - Posición del cursor

- Apariencia
- Modificar el cursor de clase
- El mensaje WM\_SETCURSOR
- Ocultar y mostrar
- Confinar el cursor
- Destrucción de cursores
- Ejemplo 33
- 33 El ratón
  - Capturar el ratón
  - Configuración
  - Mensajes
    - Mensajes del área de cliente
    - Mensajes del área de no cliente
    - Mensaje WM\_NCHITTEST
    - Mensaje WM\_MOUSEACTIVATE
  - Otros mensajes de ratón
    - Mensaje WM\_MOUSEWHEEL (Windows NT)
  - Trazar eventos del ratón (Windows NT)
    - Mensaje WM\_MOUSELEAVE (Windows NT)
    - Mensaje WM\_MOUSEHOVER (Windows NT)
  - Ejemplo 34
  - Arrastrar objetos
  - Ejemplo 35
- 34 El teclado
  - El Foco del teclado
  - Ventanas inhibidas
  - Ejemplo 36
  - Mensajes de pulsación de teclas
  - Nombres de teclas
  - El bucle de mensajes
  - Ejemplo 37
  - Mensajes de carácter
    - Teclas muertas
  - Estado de teclas
  - Ejemplo 38
  - Hot keys
  - Ejemplo 39



- Códigos de teclas virtuales
- 35 Cadenas
  - Recursos de cadenas
    - Fichero de recursos
    - Cargar cadenas desde recursos
  - Funciones para cadenas
  - Ejemplo 40
- 36 Aceleradores
  - Recursos de aceleradores
    - Fichero de recursos
    - Cargar aceleradores desde recursos
  - Bucle de mensajes para usar aceleradores
  - Crear tablas de aceleradores sin usar recursos
  - Combinar aceleradores y menús
  - Aceleradores globales
  - Diferencia entre acelerador y menú
  - Ejemplo 41
- 37 Menús 2
  - Marcas en menú
    - Menús como checkboxes
    - Menús como radiobuttons
  - Ejemplo 42
  - Inhibir y oscurecer ítems
  - Ejemplo 43
  - Más sobre ficheros de recursos
    - Sentencia MENUITEM y POPUP
    - Detalles sobre cadenas de ítems
    - Sentencia MENUEX
    - Ítems marcados y no marcados
    - Ítems activos, inactivos u oscurecidos
    - Separadores y líneas de ruptura
    - Cargar recursos
  - Ítems por defecto
  - Ejemplo 44
  - Menús flotantes o contextuales
  - Ejemplo 45
  - Acceso por teclado

- Mnemónicos
  - Acceso de teclado estándar
  - Aceleradores
- Modificar menús
- El menú de sistema
  - Modificar el menú de sistema
- Ejemplo 46
- Destrucción de menús
- Mensajes de menú
- Mapas de bits en ítems de menú
  - Modificar mapas de bits de check
  - Ítems de mapas de bits
- Ejemplo 47
- 38 La memoria
  - Memoria virtual
  - Un poco de historia
    - Memoria local y global
    - Otros atributos de la memoria en Windows
    - Objetos móviles y fijos
    - Objetos descartables y no descartables
    - Funciones *clásicas* para manejo de memoria
    - Desventajas de este modelo de memoria
  - Funciones para manejo de memoria virtual
    - Reservar direcciones de memoria virtual
    - Liberar direcciones de memoria virtual
    - Bloquear páginas de memoria asignada
    - Establecer atributos de protección de acceso
    - Obtener información sobre páginas de memoria
  - Ejemplo 48
- 39 Control edit avanzado
  - Insertar controles edit durante la ejecución
  - Cambiar la fuente de un control edit
  - Cambiar los colores de un control edit
  - Ejemplo 49
  - Controles edit de sólo lectura
  - Ejemplo 50
  - Leer contraseñas

- Ejemplo 51
- Mayúsculas y minúsculas
- Ejemplo 52
- Mensajes de notificación
  - Modificación
  - Actualización
  - Falta espacio
  - Desplazamiento horizontal y vertical
  - Pérdida y recuperación de foco
  - Texto máximo
- El buffer de texto
- Controles multilínea
  - Iniciar controles multilínea
  - Mensajes para controles multilínea
- Ejemplo 53
- Operaciones sobre selecciones de texto
- Deshacer cambios (undo)
- Modificación del texto
- Márgenes y tabuladores
- Desplazar texto
- Ejemplo 54
- Caracteres y posiciones
- Ejemplo 55
- 40 Control list box avanzado
  - Insertar controles list box durante la ejecución
  - Cambiar la fuente de un control list box
  - Cambiar los colores de un control list box
  - Ejemplo 56
  - Mensajes de notificación
    - Doble clic
    - Falta espacio
    - Pérdida y recuperación de foco
    - Selección y deselección
  - Mensajes más comunes
  - Ejemplo 57
  - El dato del ítem
  - Ejemplo 58

- Funciones para ficheros y directorios
- Ejemplo 59
- Listbox de selección sencilla y múltiple
  - Selecciones
  - Mensajes especiales para list box de selección extendida
- Ejemplo 60
- List box sin selección
- List box multicolumna
- Ejemplo 61
- Paradas de tabulación
- Ejemplo 62
- Actualizaciones de gran número de ítems
  - Optimizar la memoria
  - Optimizar el tiempo
- Ejemplo 63
- Responder al teclado
- Ejemplo 64
- Aspectos gráficos del list box
  - Ajustar la anchura de un list box
  - Ajustar la altura de los ítems
  - Ítems y coordenadas
- Ejemplo 65
- Localizaciones
- Ejemplo 66
- Otros estilos
- List box a medida (owner-draw)
- Estilos owner-draw para list box
  - List box owner-draw de altura fija
  - List box owner-draw de altura variable
  - Dibujar cada ítem
  - El mensaje WM\_DELETEITEM
- Ejemplo 67
- Otros mensajes para list box con estilos owner-draw
- Definición del orden
- 41 Control button avanzado
  - Insertar botones durante la ejecución

- Cambiar fuente
- Cambiar colores
- Modificar el bucle de mensajes
- Botones con iconos o mapas de bits
- Otros estilos para botones
  - Alineación de contenidos
  - Check box y Radio buttons
- Ejemplo 68
- Mensajes de notificación
  - Selección
  - Doble clic
  - Pérdida y recuperación de foco
  - Inhibir mensajes de notificación
- Estilos de cada tipo de botón
  - Botones pulsables
  - Check boxes
  - Radio buttons
  - Cajas de grupo
  - Botones owner-draw
- Estados de un botón
  - Selección de un botón
  - Cambios de estado
- Funciones para controles botón
  - Funciones propias de controles botón
- Modificar el estilo de un botón
- Botones owner-draw
- Ejemplo 69
- 42 Control estático avanzado
  - Insertar controles estáticos durante la ejecución
  - Cambiar fuente
  - Cambiar colores
  - Estilos estáticos gráficos
    - Marcos
    - Rectángulos
    - Ranurados
    - Ejemplos
    - Más sobre los ranurados

- Estilos estáticos de texto
- Imágenes
  - Mensajes para asignar imágenes
  - Modificadores de estilo
- Modificador de hundido
- Mensajes de notificación
- Controles estáticos owner-draw
- Ejemplo 70
- 43 Control combo box avanzado
  - Tipos de combo boxes
  - Insertar controles combo box durante la ejecución
  - Cambiar la fuente de un control combo box
  - Cambiar colores en combo box
  - Mensajes de notificación
    - Cambio en selección de lista
    - Validar selección
    - Despliegue de lista
    - Doble clic
    - Falta espacio
    - Modificación
    - Actualización
    - Pérdida y recuperación de foco
  - Otros estilos para combo box
    - Estilos para la parte de edición
    - Estilos para la lista
  - Ejemplo 71
  - Mensajes correspondientes a la lista
    - Añadir ítems
    - Recuperar información
    - Cambiar la selección
    - Buscar ítems
    - Borrar ítems
    - Otros mensajes
  - Ejemplo 72
  - El dato del ítem
  - Interfaces de usuario
  - Funciones para ficheros y directorios

- Juegos de caracteres
  - Procesar CBN\_CLOSEUP
- Selección actual
- Ejemplo 73
- El control de edición
- Actualizaciones de gran número de ítems
  - Optimizar la memoria
  - Optimizar el tiempo
- Aspectos gráficos del combo box
  - Ajustar la anchura de un combo box
  - Ajustar la altura de los ítems
- Localizaciones
- Combo boxes owner draw
  - Combo box owner-draw de altura fija
  - Combo box owner-draw de altura variable
  - Dibujar cada ítem
- Otros mensajes para combo box con estilos owner-draw
  - El mensaje WM\_DELETEITEM
- Dimensiones de la lista desplegable
- Definición del orden
- Ejemplo 74
- 44 Control scrollbar avanzado
  - Controles de barra de desplazamiento y barras estándar
  - Insertar controles scrollbar durante la ejecución
  - Cambiar colores
  - Estilos de scrollbar
    - Estilos de orientación
    - Alineamiento con los bordes
    - Opciones para cajas de tamaño
    - Alineamiento de cajas de tamaño
  - Mostrar u ocultar barras de desplazamiento
  - Deshabilitar o habilitar un control de barra de desplazamiento
  - Deshabilitar o habilitar flechas
    - Usando funciones
    - Usando mensajes
  - Mensajes de barras de desplazamiento

- Respuesta al teclado
- Ejemplo 75
- Desplazar contenido de ventanas
- Colores y medidas
  - Valores de medidas del sistema
- Otros mensajes
- Ejemplo 76
- 45 Capítulo 45 La impresora
  - Proceso de impresión
    - El spooler de impresión (print spooler)
    - El procesador de impresión (print processor)
    - La máquina de gráficos (graphics engine)
    - El monitor
  - Obtener una lista de impresoras
    - Ejemplo 77
  - Contexto de dispositivo
    - Usando CreateDC
    - Ejemplo 78
    - Usando PrintDlg
  - Ejemplo 79
- 46 Capítulo 46 Controles comunes
- 47 Capítulo 47 Control animación
  - Ficheros de recursos
  - Insertar durante la ejecución
  - Manipular la animación
    - Abrir animación
    - Reproducir
    - Detener
    - Mostrar un fotograma
    - Verificar reproducción
    - Cerrar animación
  - Mensajes de notificación
  - Ejemplo 80
- 48 Capítulo 48 Listas de imágenes
  - Crear una lista de imágenes
  - Añadir y eliminar imágenes
  - Crear listas con imágenes



- Obtener iconos
  - Mostrar imágenes
  - El color de fondo
  - Imágenes superpuestas
  - Ejemplo 81
  - Arrastre de imágenes
    - Inicio del arrastre
    - Arrastre
    - Final del arrastre
  - Ejemplo82
  - Información de imagen
- 49 Capítulo 49 Ventana de estado
  - Cómo crear ventanas de estado
  - Estilos
  - Ayuda para menús
  - Ejemplo 83
  - Tamaño y altura
  - Ventanas de estado con varias partes
  - Manejar texto
  - Ejemplo 84
  - Ventanas de estado owner-draw
  - Ejemplo 85
  - Ventanas de estado simples
- 50 Capítulo 50 Barra de progreso
  - Estilos visuales
    - Fichero de manifiesto
    - Manifiesto en fichero de recursos
  - Estilos
  - Cómo crear barras de progreso
  - Rangos
  - Posicion
  - Colores
  - Ejemplo 86
- 51 Capítulo 51 Control Tooltip
  - Creación de tooltip
  - Estilos
  - Activar y desactivar tooltips

- Cambios de color
- Asignar título e icono
- Limitar anchura
- Asignar a herramienta
  - Asignar tooltip a un control
  - Asignar tooltip a un rectángulo
  - Eliminar un control de un tooltip
  - Usar cadenas de recursos
- Ejemplo 87
- Notificaciones
  - Mensaje de petición de texto
- Ejemplo 88
- Notificaciones de mostrar y ocultar
- Personalización
- Ejemplo 89
- Otros mensajes
- 52 Capítulo 52 Control UpDown
  - Creación de un control UpDown
  - Especificar una ventana amiga
  - Estilos
  - Rango y posición actual
  - Ficheros de recursos
  - Aceleradores
  - Bases de numeración
  - Mensajes de notificación
  - Ejemplo 90
- 53 Control de cabecera
  - Creación de un control de cabecera
  - Añadir columnas
  - Cambio de tamaño de la ventana padre
  - Estilos
  - Mensajes de gestión de columnas
  - Mensajes relacionados con el orden de columnas
  - Mensajes de arrastre de items
  - Arrastre de divisores
  - Mensajes de filtros
  - Indicativos de orden

- Mensajes de foco de teclado
- Mensajes de situación en ventana
- Botón de desplegar
- Cajas de chequeo
- Overflow
- Mensajes de gestión de mapas de bits
- Mensajes de codificación de caracteres
- Acción del ratón sobre items
- Notificaciones de modificación de item
- Pulsaciones de tecla
- Inserción con datos incompletos
- Otros mensajes de notificación
- Ejemplo 91
- Ejemplo 92
- Ejemplo 93
- 54 Control ComboBoxEx
  - Insertar durante la ejecución
  - Estilos
  - Lista de imágenes
  - Insertar items
  - Modificar un item
  - Obtener información de un item
  - Eliminar un item
  - Edición de valores
  - Ejemplo 94
  - Ficheros de recursos
  - Mensajes de formato de caracteres
  - Controles base
  - Operaciones de arrastre
  - Temas de Windows
  - Ejemplo 95
- 55 Control de selección de fecha y hora
  - Insertar durante la ejecución
  - Desde fichero de recursos
  - Estilos
  - Asignar un valor
  - Obtener un valor

- Establecer rangos
- Obtener rangos
- Atributos del calendario mensual
  - Obtener el manipulador de ventana
  - Cambiar la fuente
  - Cambio de estilos
  - Cambio de colores
  - Cerrar calendario
- Asignar formato
- Calcular el tamaño del control
- Obtener información
- Códigos de notificación
  - Notificación de cambio de fecha y hora
  - Control de calendario mensual desplegado
  - Campos de retrollamada
  - Cadenas de usuario
- Ejemplo 96
- Ejemplo 97
- 56 Control de calendario
  - Insertar durante la ejecución
  - Desde fichero de recursos
  - Estilos
  - Control de calendario de selección simple
    - Seleccionar fecha
    - Obtener fecha seleccionada
  - Selección múltiple
    - Asignación de varias fechas
    - Obtener asignación múltiple
    - Rango máximo de selección
    - Selección fuera de la vista
  - Fechas seleccionables
  - Aspecto gráfico
    - Borde
    - Colores
    - Estado de días
  - Calendarios contenidos
  - Obtener información

- Rangos visibles
    - Medidas de la cadena hoy
  - Tipos de calendario
  - Navegacion
  - Hoy
  - Vistas
  - Formato de juego de caracteres
  - Puntos de prueba
  - Notificaciones
    - Cambio de selección
    - Selección
    - Cambio de vista
    - Liberación de captura de ratón
  - Ejemplo 98
- 57 Control Trackbar
  - Insertar durante la ejecución
  - Insertar desde fichero de recursos
  - Partes del control trackbar
  - Establecer rango de valores
  - Modificar y leer posición del deslizador
  - Estilos
    - Que afectan a la orientación
    - Obtener marcas
    - Relacionados con los rangos
    - Relacionados con el deslizador
    - Eliminar marcas
    - Posiciones de marcas
    - Relacionados con los rangos
    - Relacionados con el deslizador
  - Tooltips
  - Desplazamientos
  - Ventanas compañeras
  - Delimitadores de áreas
  - Formato de caracteres
  - Trackbar custom draw
  - Ejemplo 99
- A Glosario