Programación ActiveX con Visual Basic

8 Temas de interés

Antes de adentrarnos en la sección del libro dedicada exclusivamente al desarrollo de controles ActiveX, no está de más repasar algunos temas de interés relacionados con ciertas novedades incluidas en la versión 5.0 de Visual Basic. Quizá ya conozca estos temas, pero en cualquier caso nunca está de más dar un pequeño repaso.

Algunos de los puntos que se van a estudiar ya se han usado en capítulos previos, como es el caso de las enumeraciones, mientras que otros son nuevos. Veamos detalladamente cómo aprovecharlos y utilizarlos en la creación de nuestros propios componentes.

Enumeraciones

Ya hemos usado enumeraciones en algunos de los ejemplos desarrollados en capítulos previos, aunque sin detenernos a estudiar su sintaxis y funcionamiento. La declaración de una enumeración es análoga a la de un tipo definido por un usuario, si bien en este caso los miembros son sólo constantes representativas de los valores que podría tomar una variable declarada de ese tipo. La siguiente es la definición típica de una enumeración:

```
Public Enum DiaSemana
        Lunes
        Martes
        Miercoles
        Jueves
        Viernes
        Sabado
        Domingo
End Enum

Dim Dia As DiaSemana
```

La variable **Dia** es del tipo **DiaSemana**, una enumeración cuyos valores se representan mediante las constantes **Lunes**, **Martes**, ... **Domingo**. En realidad, internamente **Dia** es una variable de tipo *Long*, por lo que sería posible asignarle cualquier valor entero que estuviese en el rango de este tipo numérico. Una asignación como la siguiente, por ejemplo, no generaría error alguno:

```
Dia = 12
```

Al definir una enumeración es posible asignar valores explícitamente a las constantes que la componen. Para ello bastará con disponer detrás del nombre de la constante el operador = y el valor a asignar. En la siguiente enumeración, por ejemplo, se asigna de forma explícita un valor diferente a cada una de la constantes.

```
Public Enum Color
        Rojo = vbRed
        Azul = vbBlue
        Verde = vbGreen
End Enum
```

Si tras asignar un valor a una constante se omite el valor de las siguientes, Visual Basic les asignará automáticamente el valor de la constante anterior incrementado en una unidad.

Por supuesto las enumeraciones pueden ser privadas, en cuyo caso su uso se restringirá sólo al interior del módulo en que se haya declarado.

Ventajas de usar enumeraciones

Si, como se acaba de decir, es posible asignar cualquier valor a una variable cuyo tipo es una enumeración, ¿qué ventajas tiene usarlas? ¿No sería lo mismo declarar la variable de tipo *Long* y usar constantes definidas en la forma tradicional, con la palabra clave *Const*? Si el uso de la variable y las constantes va a ser interno a nuestro componente, lo cierto es que no habrá mucha diferencia entre usar un método u otro, si exceptuamos que la declaración de una variable con el tipo **DiaSemana** define mejor los valores que puede almacenar. Pero si, por el contrario, la variable va a ser usada como

almacenamiento de una propiedad o tipo para los métodos de acceso correspondientes, las enumeraciones tienen visibles ventajas.

Si definimos una propiedad con el tipo *Long*, durante el diseño será posible asignarle cualquier valor usando el Inspector de propiedades. Por el contrario, si definimos una enumeración y la usamos como tipo, el Inspector de objetos mostrará una lista desplegable con los valores posibles, no permitiendo al usuario del control que asigne valores inválidos. Cuando en el Editor de código se haga referencia a la propiedad para asignarle un valor, Visual Basic, en caso de que sea él el contenedor del control, mostrará automáticamente una lista desplegable con la enumeración de valores. Lógicamente será posible asignar cualquier otro tecleándolo, ahí es donde el método de asignación de la propiedad escrito por nosotros entra en acción, rechazando cualquier valor no válido.

A pesar de que sólo lo dicho hasta ahora ya justificaría el uso de las enumeraciones en sustitución de constantes independientes, hay ventajas adicionales al usar esta nueva característica del lenguaje añadida en la versión 5.0. Las constantes definidas en un módulo de clase que forma parte de un componente ActiveX, son sustituidas automáticamente por los valores correspondientes. Es decir, las constantes en sí no existirán en el componente, son sólo un recurso para facilitar la codificación. Una enumeración, por el contrario, pasará a formar parte del componente ActiveX, de tal forma que leyendo su librería de tipos será posible recuperar la información relativa a la enumeración. En realidad es este hecho, que la enumeración se encuentre en la librería de tipos ActiveX, el que permite que Visual Basic u otro contenedor muestre los valores correctos en tiempo de diseño, en el Inspector de propiedades o un elemento equivalente en otro entorno.

Por lo tanto, siempre que vaya a definir una propiedad que pueda tomar varios valores numéricos asociados a un rango discreto de posibilidades, en lugar de usar el tipo *Long* o el tipo *Integer* defina una enumeración y úsela como tipo. Los beneficios obtenidos justifican este trabajo adicional.

Colecciones

La colección no es un elemento nuevo en Visual Basic, aunque su funcionalidad se ha ido ampliando con cada nueva versión. Inicialmente existían colecciones predefinidas que se podían usar, por ejemplo, para acceder a los formularios o controles. Posteriormente, al aparecer el objeto *Collection*, se abrió la puerta al uso de las colecciones para almacenar cualquier serie de datos que pudiera sernos útil sin las limitaciones propias de las matrices. A partir de la versión 5.0 se facilita la creación de colecciones propias, con una funcionalidad idéntica a la del propio objeto *Collection*.

Seguramente estará pensando que ya en versiones anteriores de Visual Basic, como la 4.0, usted podía crear sus propias clases de colecciones. Es cierto, pero existían limitaciones. Una de las ventajas del uso de las colecciones es que es posible recorrerlas sin necesidad de conocer el número de elementos existentes ni usar índices, gracias a la construcción *For Each* ***objeto*** *In*. Esta modalidad de bucle, sin embargo, no funciona con ningún otro objeto, sólo con el tipo *Collection*. En la versión 5.0, no obstante, sí es posible conseguir la misma funcionalidad en cualquier otra clase de colección creada por nosotros.

Objetos *Collection* públicos

Si está diseñando un componente que cuenta con una colección de objetos cualesquiera y, por flexibilidad, quiere que los usuarios puedan acceder a esa colección, accediendo a sus elementos, añadiendo o eliminando, una de las posibilidades que tiene es hacer público el objeto *Collection*. Para ello bastaría con declarar una variable pública del tipo *Collection*, creando la colección directamente en la propia declaración o bien en un evento de inicialización.

Este método, no cabe duda, es el más sencillo posible. Además es muy flexible, puesto que el usuario de nuestro componente podrá acceder a la colección sin limitación alguna, pudiendo acceder a los elementos y manipularlos. Esta flexibilidad, sin embargo, es un arma de doble filo, puesto que el usuario podría insertar en la colección objetos de tipos no válidos para nosotros o valores inapropiados. No podemos controlar esto, puesto que los métodos que facilitan las operaciones sobre la colección son propios del objeto *Collection*, no nuestros.

Clases de colecciones propias

Para evitar los problemas que se derivan de hacer público un miembro de datos de un objeto, lo mejor es crear una clase implementando las propiedades y métodos que se precisen. De esta forma, mediante los métodos de acceso correspondientes, podremos evitar que el usuario introduzca valores erróneos en la colección.

Con el fin de mantener la coherencia con cualquier otra colección, nuestra clase deberá contar con los métodos *Add* y *Remove*, así como con las propiedades *Count* e *Item*, siendo ésta última la propiedad predeterminada. De esta forma podríamos crear, por ejemplo, una colección que tan sólo aceptase días de la semana, tal y como se muestra a

continuación.

```
Public Enum DiaSemana ' Enumeración de valores posibles
    Lunes
    Martes
    Miercoles
    Jueves
    Viernes
    Sabado
    Domingo
End Enum

    ' Objeto privado para mantener la colección
Private PColeccion As New Collection

    ' Propiedad sólo de lectura que devuelve el número de elementos
Public Property Get Count() As Long
    Count = PColeccion.Count
End Property

    ' Propiedad sólo de lectura que devuelve un elemento
Public Property Get Item(Indice As Long) As DiaSemana
    If Indice > 0 And Indice <= PColeccion.Count Then _
        Item = PColeccion(Indice)
End Property

    ' Método para añadir un nuevo elemento
Public Sub Add(Elemento As DiaSemana)
    If Elemento >= Lunes And Elemento <= Domingo Then _
        PColeccion.Add Elemento
End Sub

    ' Método para eliminar un elemento existentes
Public Sub Remove(Indice As Long)
    If Indice > 0 And Indice <= PColeccion.Count Then _
        PColeccion.Remove Indice
End Sub
```

Suponiendo que este código se aloja en un módulo de clase llamado **PAXDiasSemana**, podríamos crear un objeto de dicha clase y usar sus propiedades y métodos para añadir, examinar y eliminar elementos. Lógicamente también podríamos usar la clase como tipo de propiedad de un control, teniendo la seguridad de que el usuario no va a poder introducir en la colección valores no válidos. El control **PAXControl** es un ejemplo simple que implementa una propiedad, llamada **Dias**, que facilita el acceso a la colección privada **PDias**, como puede verse a continuación.

```
     ' Variable privada para almacenar la colección
Private PDias As New PAXDiasSemana

    ' Método de lectura de la propiedad Dias
Public Property Get Dias() As PAXDiasSemana
    Set Dias = PDias
End Property
```

Este control, lógicamente, no es de suma utilidad. Insertando uno en un formulario podríamos añadir, eliminar y acceder a los elementos de la propiedad **Dias** como si de cualquier otra colección se tratase. Sin embargo, al intentar usar la construcción *For Each* obtendríamos un error.

```
        ' Al pulsar el botón
Private Sub Command1_Click()
    With PAXControl1
        .Dias.Add Lunes ' Añadimos dos elementos
        .Dias.Add Viernes
        .Dias.Remove 1 ' eliminamos uno
        Cls
```

```
        Print "Contador = " & .Dias.Count ' mostramos el número de elementos
        Dim Dia As Variant
        For Each Dia In .Dias
            Print Dia ' y el contenido de cada uno
        Next
    End With
End Sub
```

La interfaz *IEnumVARIANT*

¿Qué tiene de especial el tipo *Collection*? ¿Por qué se puede usar con la construcción *For Each*? La respuesta está en que, a diferencia de cualquier clase que nosotros podamos definir, *Collection* implementa, entre otras, la interfaz *IEnumVARIANT*. Esta es una interfaz estándar COM que facilita la enumeración de valores por parte de un objeto, sin importar su tipo puesto que se usa siempre el tipo *Variant*.

La instrucción *For Each* lo que hace es obtener la interfaz *IEnumVARIANT* del objeto que se facilita a continuación, llamando para ello al método oculto *NewEnum*. Sólo los objetos *Collection* cuentan con este método, pero está claro que nosotros podemos añadirle uno a nuestra propia clase. El problema estriba en que nuestra propia clase no implementa la interfaz *IEnumVARIANT*, por lo que de implementar el método *NewEnum* ¿qué devolveríamos?

Todo objeto *Collection* cuenta con el método *NewEnum*, que al estar oculto recibe el nombre *_NewEnum*. Podemos delegar nuestro método *NewEnum* en el de la colección interna que estamos usando, al igual que hemos delegado muchas funciones en el objeto *UserControl* en capítulos previos. Para ello bastará con que nuestro método *NewEnum* devuelva el valor devuelto a su vez por el método *NewEnum* de dicha colección.

Como se acaba de decir, el método *NewEnum* del objeto *Collection* está oculto, por lo que su nombre es *_NewEnum*. Ese subrayado inicial impide que podamos, en Visual Basic, referirnos directamente al identificador. No podemos escribir una referencia como la siguiente:

```
Set NewEnum = PColeccion._NewEnum
```

Para poder usar estos caracteres inválidos en un identificador, es necesario delimitar éste mediante unos corchetes. De esta forma, nuestro método *NewEnum* podría quedar de la siguiente forma.

```
        ' Devuelve la interfaz que permite recorrer la enumeración
Public Property Get NewEnum() As IEnumVARIANT
    Set NewEnum = PColeccion.[_NewEnum]
End Property
```

Escrito este método deberemos usar la ventana de atributos de procedimiento para establecer algunos parámetros. Nuestro método ha de ser oculto, por lo que hay que activar la opción **Ocultar este miembro**, tal y como se puede apreciar en la figura 8.1. Además tendremos que asignar el identificador **-4** al método, ya que *For Each* no usa un método llamado *NewEnum*, sino un método con ese identificador.

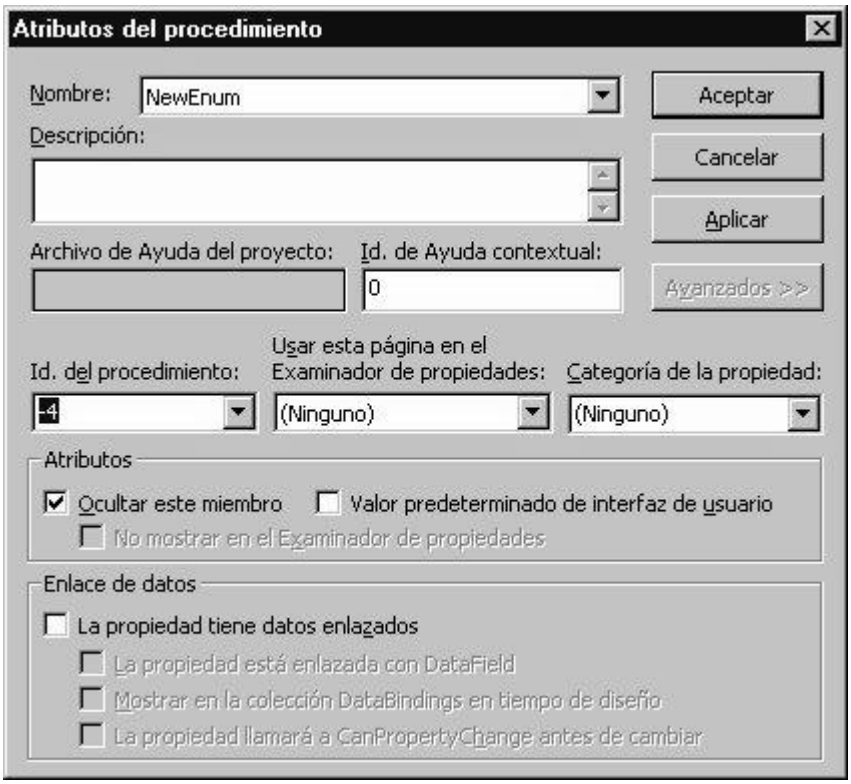

Figura 8.1: Atributos del método *NewEnum*

Dados todos estos pasos ya podremos usar una colección **PAXDiasSemana** como cualquier otra colección, siendo posible enumerar sus elementos mediante la construcción *For Each*. El proceso es algo "retorcido", por decirlo de alguna forma, pero no tenemos otra opción para conseguir el resultado que queremos, a no ser que deseemos implementar nosotros directamente la interfaz *IEnumVARIANT*.

Direcciones de funciones

A pesar de que la posibilidad de acceder a las funciones de la API de Windows desde Visual Basic siempre ha estado presente, lo cierto es que hasta la versión 5.0 no se ha incorporado al lenguaje un elemento "fundamental" para muchos programadores, como es la posibilidad de obtener la dirección de una función. ¿Para qué necesitamos esta dirección? Existen multitud de casos en los que puede ser imprescindible. Cualquier función Windows que requiera el uso de una función *callback*, por ejemplo, precisará la dirección de dicha función. Asimismo será imprescindible obtener la dirección de una función si queremos, por ejemplo, subclasificar una ventana.

Una de las novedades de la versión 5.0 de Visual Basic es el operador *AddressOf*, que dispuesto delante del nombre de una función devuelve su dirección. Esta operación, aparentemente tan simple, abre la puerta para que los programadores Visual Basic puedan efectuar muchas operaciones que hasta ahora estaban fuera de su alcance.

Limitaciones de *AddressOf*

El uso de punteros a funciones es potencialmente muy peligroso, los programadores de lenguajes como C/C++ y Pascal lo saben bien. Es muy fácil asignar una dirección errónea y bloquear la aplicación o, en ocasiones, incluso el sistema completo. Seguramente esta es la razón de que el operador *AddressOf* esté tan limitado, a pesar de lo cual, como se ha dicho antes, puede ser muy útil.

Este operador tan sólo puede ser usado como parámetro en la llamada a un método o función, es decir, no se puede almacenar el valor devuelto por *AddressOf* directamente en una variable, puesto que una sentencia como la siguiente es inválida.

```
Dim Direccion As Long

Direccion = AddressOf MiFuncion
```

Tras escribir la sentencia de asignación y pulsar <Intro> verá que automáticamente Visual Basic avisa que la expresión

no es válida. Si queremos almacenar una dirección en una variable, tendremos que escribir una función como la siguiente:

```
 Function DireccionFuncion(Dir As Long) As Long
    DireccionFuncion = Dir
End Function

....
Direccion = DireccionFuncion(AddressOf Enumera.EnumeraVentanas)
```

En este caso la variable **Direccion** sí almacena la dirección de la función, que ha sido pasada como parámetro a la función **DireccionFuncion**. Esta función la devuelve sin más, no es necesario realizar operación alguna.

La función que se utiliza como parámetro del operador *AddressOf* ha de alojarse siempre en un módulo de código estándar. No es posible obtener la dirección de un método de un objeto, por ejemplo un control, puesto que un elemento como ese no es una función "normal", sino una entrada en una tabla de direcciones de una interfaz COM, tal y como vimos en el capítulo correspondiente.

Este hecho, que la función cuya dirección se va a obtener deba encontrarse en un módulo estándar, complica su uso en proyectos como controles y componentes ActiveX, en los cuales la encapsulación es un elemento de vital importancia.

Un control enumerador de ventanas

Veamos con un pequeño ejemplo cómo podemos combinar un módulo estándar con un control de usuario. La finalidad será construir un control, no visible durante la ejecución, que contará con un método y un evento. El método, al que llamaremos **EnumeraVentanas**, se encargará de obtener el nombre de todas las ventanas de primer nivel que haya abiertas, generando un evento **Ventana** por cada una de ellas. El método asociado a ese evento recibirá como parámetro el título de la ventana.

Para obtener una lista de las ventanas podemos usar distintas funciones de la API de Windows. En este caso vamos a utilizar *EnumWindows*, función a la que hay que pasar dos parámetros: la dirección de una función *callback* y un dato de usuario que puede ser cualquier información que nos interese pasar a esa función *callback*. Tras llamar a *EnumWindows*, se pondrá en marcha un proceso por el que Windows llamará a la función cuya dirección se ha pasado tantas veces como ventanas haya. En cada llamada se facilita como primer parámetro el identificador de una ventana y, como segundo, el dato que nosotros mismos hubiésemos pasado en la llamada a *EnumWindows*.

Teniendo un identificador de ventana, para recuperar su nombre usaremos la función *GetWindowText*. Al llamarla entregaremos como primer parámetro el identificador, como segundo una cadena y como tercero un entero indicando el espacio disponible en dicha cadena. Esta función devolverá como resultado un número comunicando la longitud de la cadena devuelta, o cero en caso de que la ventana no tenga nombre o no exista.

Puesto que la función *callback* tendrá que encontrarse en un módulo estándar, deberemos establecer de alguna forma una comunicación entre ella y el control que vamos a diseñar. Para ello declararemos en dicho módulo una variable que nos permitirá mantener una referencia al control que lo está usando. Cuando se cree un **PAXEnumeraVentanas** él mismo llamará al método **Asocia** que hay en el módulo, facilitando como parámetro una referencia a sí mismo. En el momento en que el control se vaya a destruir, se llamará al método **Cancela** para eliminar la asociación. El código del control es el que se puede ver a continuación.

```
 Option Explicit

    ' Evento que generará el control
Event Ventana(Titulo As String)

    ' Este método público será llamado desde el código del módulo estándar
Public Sub TituloVentana(Titulo As String)
    RaiseEvent Ventana(Titulo)
End Sub

    ' Este es el único método accesible para el usuario
Public Sub EnumeraVentanas()
    Enumera.ListaVentanas
End Sub
```

```
' Al crear el control
Private Sub UserControl_Initialize()
    Enumera.Asocia Me ' lo asociamos con el módulo
End Sub

    ' Al destruir el control
Private Sub UserControl_Terminate()
    Enumera.Cancela ' cancelamos la asociación
End Sub
```

El método público **TituloVentana** está pensado para ser llamado desde el módulo de código estándar cada vez que se obtenga un nombre de ventana. Sin embargo, este método no debería de estar accesible para el usuario final, por lo que usando la ventana de propiedades de procedimiento lo haremos oculto. Que el método no aparezca en el Examinador de objetos ni en la lista de miembros cuando se está escribiendo código, no significa que no se le pueda llamar. En el código del módulo estándar, mostrado seguidamente, se llama a **TituloVentana** por cada título de ventana que se recupera. El control traduce esta llamada en un evento **Ventana**.

```
     ' Control asociado a la operación
Private Objeto As PAXEnumeraVentanas

    ' Esta es la función callback, que recibe el identificador de la
    ' ventana y un parámetro adicional
Private Function EnumeraVentanas(ByVal hwnd As Long, ByVal LPARAM As Long) As Lon
    Dim Titulo As String * 256
        ' si podemos obtener el título
    If GetWindowText(hwnd, Titulo, 255) <> 0 Then _
        Objeto.TituloVentana Titulo ' lo pasamos al control
    EnumeraVentanas = True ' y seguimos con la enumeración
End Function

    ' Al llamar a este procedimiento
Public Sub ListaVentanas()
    If TypeName(Objeto) = "PAXEnumeraVentanas" Then _
        EnumWindows AddressOf EnumeraVentanas, 0 ' e iniciamos la enumeración
End Sub

    ' Este método asocia el control con el módulo
Public Sub Asocia(L As PAXEnumeraVentanas)
    Set Objeto = L
End Sub

    ' Este método cancela la asociación
Public Sub Cancela()
    Set Objeto = Nothing
End Sub
```

Tras dar el valor *True* a la propiedad *InvisibleAtRuntime* del control, ya estamos preparados para comprobar su funcionamiento. Bastará con insertar en un formulario una lista, un botón y un **PAXEnumeraVentanas**. Al pulsar el botón se limpiará la lista y se llamará al método **EnumeraVentanas** de nuestro control. Cada vez que se reciba un evento **Ventana** se añadirá el título a la lista. La figura 8.2 muestra el programa en funcionamiento. Antes de ejecutar cualquier programa en el que se utiliza el operador *AddressOf* siempre es recomendable guardar el trabajo, ya que un fallo podría hacer que todo se perdiese.

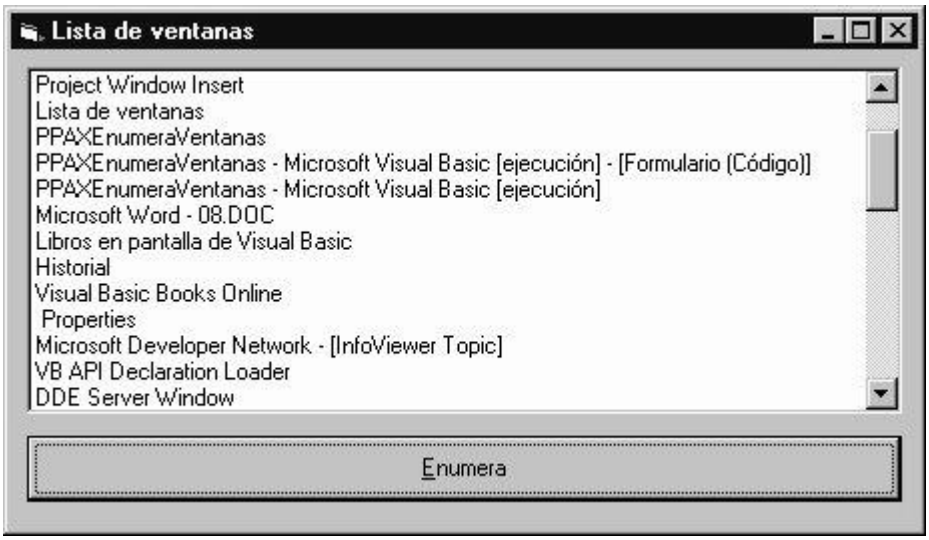

Figura 8.2: Lista de ventanas recuperada por el control **PAXEnumeraVentanas**

Observe que la función **EnumeraVentanas** devuelve siempre el valor *True*, ya que de lo contrario el proceso se detendría de forma inmediata, obteniéndose tan sólo el identificador de una ventana. Fíjese también en la comprobación que se realiza, en el procedimiento **ListaVentanas**, justo antes de llamar a *EnumWindows*. En caso de que la variable **Objeto** no tenga una referencia a un **PAXEnumeraVentanas** no se realiza el proceso de enumeración.

Los problemas de las variables públicas

Como ya sabe, las variables declaradas en el interior de un módulo de clase o un control no existen por sí solas, sino que forman parte de un objeto. De esta forma, hasta que no se crea el objeto la variable no existe. No es posible acceder directamente a la variable, sino que hay siempre que componer una referencia formada por el objeto y el nombre de la variable.

Una variable en un módulo de código estándar, a pesar de que sea una variable privada, es un espacio de memoria global. Esto quiere decir que es compartida por todas las copias de los objetos que puedan crearse, en este caso por todos los controles **PAXEnumeraVentanas**. Puede hacer una prueba muy simple: habrá el formulario del programa anterior e inserte una nueva lista y un nuevo **PAXEnumeraVentanas**, de tal forma que al pulsar el botón se inicien dos enumeraciones, dirigiendo la salida de una a una lista y la de otra a la segunda lista, tal y como se deduce del código siguiente:

```
 Private Sub Command1_Click()
    List1.Clear
    List2.Clear
    PAXEnumeraVentanas1.EnumeraVentanas
    PAXEnumeraVentanas2.EnumeraVentanas
End Sub

Private Sub Form_Resize()
    List1.Width = ScaleWidth - List1.Left * 2
    Command1.Width = List1.Width
End Sub

Private Sub PAXEnumeraVentanas1_Ventana(Titulo As String)
    List1.AddItem Titulo
End Sub

Private Sub PAXEnumeraVentanas2_Ventana(Titulo As String)
    List2.AddItem Titulo
End Sub
```

Al ejecutar el programa obtendrá un resultado similar al de la figura 8.3. En una de las listas aparecen todas las entradas duplicadas, mientras que la otra permanece vacía. ¿Por qué ocurre esto? Analicemos detenidamente el proceso que se sigue hasta llegar aquí. Al crear la primera copia del control **PAXEnumeraVentanas** éste se asocia con el módulo estándar, que guarda la referencia en la variable global **Objeto**. Acto seguido se crea la segunda copia del control, que llama al método **Asocia** para enlazarse con el módulo global. La asignación de la nueva referencia a la variable **Objeto**

hace que la anterior referencia se pierda. Cuando el primer control llama al procedimiento **EnumeraVentanas**, el destino de la notificación es ya la segunda copia del control, puesta que ésta es la referencia almacenada en la variable **Objeto**.

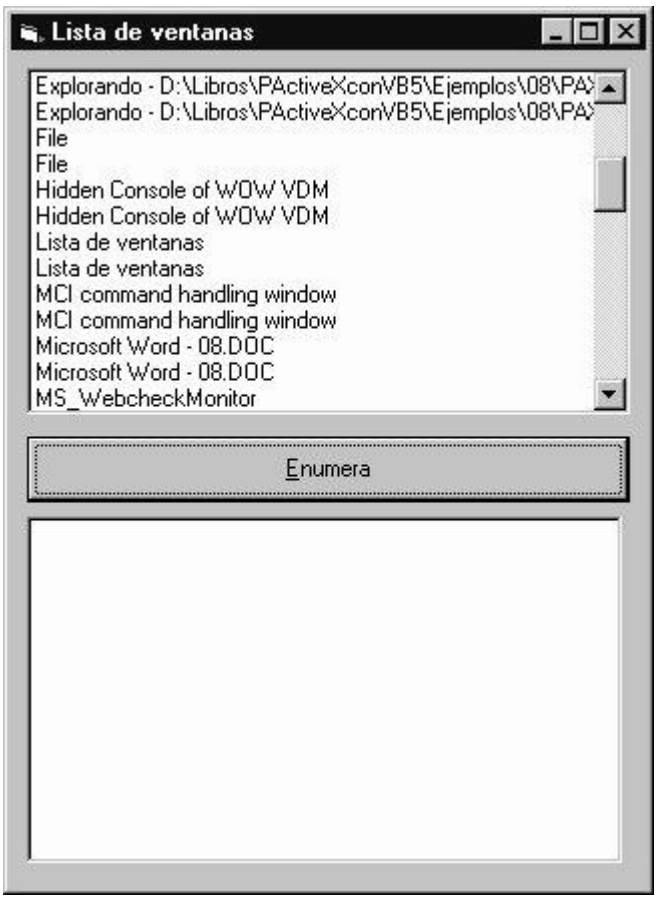

Figura 8.3: Resultado al usar dos copias del control **PAXEnumeraVentanas**

Cuando el segundo control se destruya, produciéndose el evento *Terminate*, tendrá lugar una llamada al procedimiento **Cancela** del módulo estándar. En ese momento se asigna el valor *Nothing* a la variable **Objeto**. La primera copia del **PAXEnumeraVentanas**, sin embargo, cree que aún permanece asociado, por lo que podría llamar de nuevo a **ListaVentanas**. Por suerte en ese procedimiento hemos incluido una comprobación que evita la llamada a *EnumWindows* en caso de que la referencia **Objeto** sea *Nothing*, de lo contrario se produciría un error.

Encapsulación de un módulo estándar

Para conseguir un correcto funcionamiento en cualquier caso, debemos encontrar una forma de encapsular el código del módulo estándar. Lo más fácil sería alojar ese código en un módulo de clase, pero ya hemos dicho antes que el operador *AddressOf* exige que la función cuya dirección se quiere obtener esté en un módulo estándar.

¿Cómo funciona un módulo de clase? Cada vez que se crea un objeto de una clase se asigna el espacio necesario para alojar las variables definidas a nivel de módulo, de tal forma que cada objeto, a pesar de compartir el mismo código ejecutable con los demás, dispone de su conjunto particular de variables, independientes de las de los demás objetos. Por lo tanto, de lo que se trata es de conseguir que cada objeto asociado a un módulo estándar cuente con su propia copia de las variables. ¿Cómo conseguir esto? Lo cierto es que hay muchas formas, casi tantas como podamos imaginar. En el caso que nos ocupa, sin embargo, lo más fácil es eliminar la variable global.

La función *EnumWindows* acepta como segundo parámetro un valor cualquiera, valor que será posteriormente facilitado a la función *callback*. Inicialmente dicho parámetro está definido como *Long* y para ser pasado por valor, pero nada nos impide modificar esta definición para poder pasar una referencia al control **PAXEnumeraVentanas** asociado. Esta posibilidad, de la cual disponen muchas funciones de la API de Windows, nos facilitará mucho el trabajo.

Comenzaremos por modificar el código del control, eliminando los métodos correspondientes a los eventos *Initialize* y *Terminate*. Será al llamar al procedimiento **ListaVentanas** cuando facilitemos una referencia al propio control, tal y como puede verse en el código siguiente.

```
    ' Evento que generará el control
Event Ventana(Titulo As String)

    ' Este método público será llamado desde el código del módulo estándar
Public Sub TituloVentana(Titulo As String)
    RaiseEvent Ventana(Titulo)
End Sub

    ' Este es el único método accesible para el usuario
Public Sub EnumeraVentanas()
    Enumera.ListaVentanas Me
End Sub
```

Éste es todo el código del control, que se ha simplificado apreciablemente. También el código del módulo estándar se va a reducir. Ya no son precisos los procedimientos **Asocia** y **Cancela**, que pueden ser eliminados. Tampoco necesitamos ya la variable **Objeto**, puesto que el parámetro que se pasa a **ListaVentanas** no va a almacenarse, sino que se entregará como parámetro a la función *EnumWindows*. Ésta, a su vez, nos lo devolverá en cada llamada a la función **EnumeraVentanas**, cuyo código hemos modificado para que **LPARAM** no sea de tipo *Long*, sino una referencia a un control **PAXEnumeraVentanas2**. Usando esta referencia no tenemos mas que hacer la llamada para notificar la recepción de un título de ventana.

```
 ' Para obtener una enumeración de todas las ventanas de primer nivel
Private Declare Function EnumWindows Lib "user32" _
(ByVal lpEnumFunc As Long, _
ByRef LPARAM As Any) As Long

    ' Esta es la función callback, que recibe el identificador de la ventana y un
Private Function EnumeraVentanas(ByVal hwnd As Long, _
        ByRef LPARAM As PAXEnumeraVentanas2) As Long
    Dim Titulo As String * 256
        ' si podemos obtener el título
    If GetWindowText(hwnd, Titulo, 255) <> 0 Then _
        LPARAM.TituloVentana Titulo ' lo pasamos al control
    EnumeraVentanas = True ' y seguimos con la enumeración
End Function

    ' Al llamar a este procedimiento
Public Sub ListaVentanas(L As PAXEnumeraVentanas2)
    EnumWindows AddressOf EnumeraVentanas, L ' e iniciamos la enumeración
End Sub
```

Hechas estas modificaciones podemos realizar la misma prueba anterior, usando dos controles **PAXEnumeraVentanas2** para llenar dos listas con los nombres de las ventanas. En este caso no hay ningún problema, cada lista recibe los elementos que le corresponden.

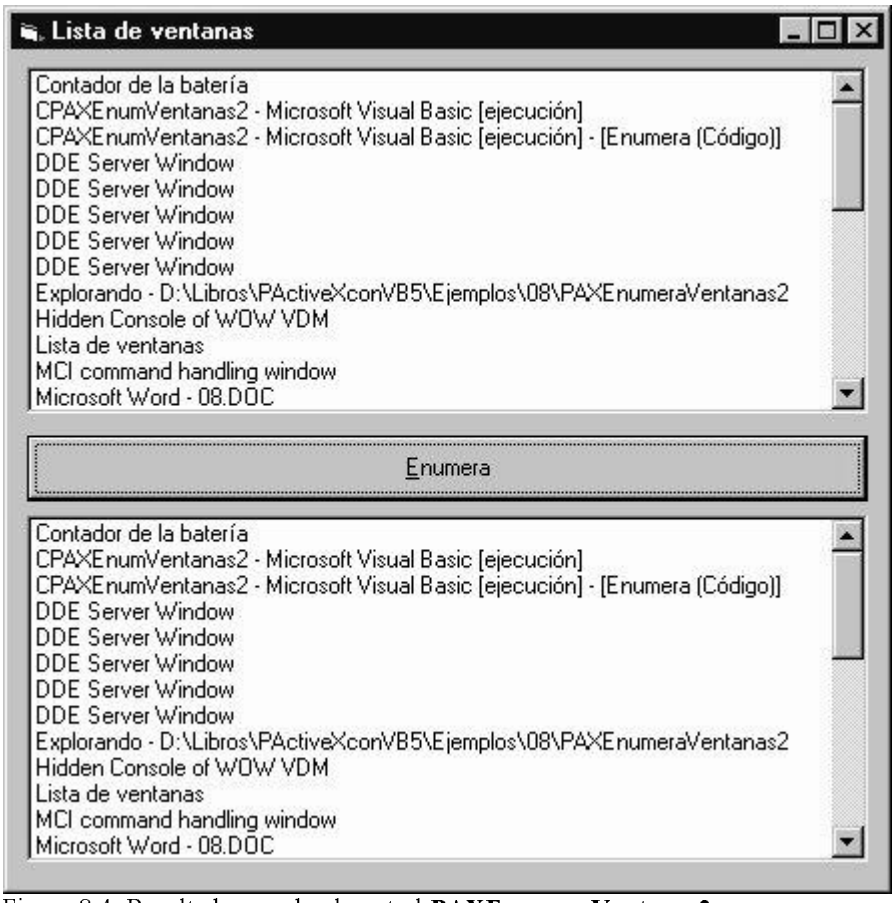

Figura 8.4: Resultado usando el control **PAXEnumeraVentanas2**

En posteriores ejemplos tendremos ocasión de conocer otras formas de encapsular el código de un módulo estándar, de tal modo que pueda ser utilizado de forma segura por cualquier número de clientes de forma simultánea.

Subclasificación

La posibilidad de obtener la dirección de una función, mediante el operador *AddressOf*, no es útil tan sólo en casos como el visto en el punto anterior, sino que nos permite realizar tareas mucho más interesantes y que, hasta ahora, estaban fuera del alcance de los programadores que utilizan Visual Basic. Una de esas tareas es la posibilidad de subclasificar una ventana, interceptando todos los mensajes que recibe. Subclasificando una ventana se pueden realizar procesos que no serían posibles de otra forma, salvo utilizando controles o librerías externas, pero también es muy fácil provocar el mal funcionamiento de un programa. Se trata, por lo tanto, de una posibilidad muy potente y peligrosa a la vez.

No sólo los formularios son ventanas

El primer concepto que debemos tener claro es qué es una ventana en el sistema operativo Windows. Si le preguntamos a cualquier usuario, posiblemente nos dirá que una ventana es ese espacio en pantalla delimitado por un borde y en cuyo interior se solicitan o muestran datos. Para la mayoría de los programadores de entornos de desarrollo rápido, como Visual Basic, una ventana es un contenedor en el que se pueden alojar componentes, por lo que una ventana sería el formulario. Para Windows, sin embargo, una ventana es una estructura de datos en memoria que define un posible aspecto, además de contener un apuntador a una función que será la encargada de gestionar los mensajes que se reciban.

Son ventanas, por lo tanto, los formularios, los botones, las listas y prácticamente cualquier otro control visual salvo excepciones. También son ventanas los controles creados por nosotros mismos. Lo único que diferencia a estos elementos es su clase, previamente registrada en Windows mediante *RegisterClass* o *RegisterClassEx*. Las clases de ventana no tienen nada que ver con las clases de objetos Visual Basic. Una clase de ventana es simplemente una estructura de datos que se registra con un cierto nombre, de tal forma que después es posible crear ventanas de esa clase usando dicho nombre.

Procedimientos de ventana

Entre los miembros de la estructura de datos que define una clase de ventana, existe uno que contiene la dirección de una

función conocida habitualmente como "procedimiento de ventana". Dicha función es compartida, por regla general, por todas las ventanas de esa clase. Cuando se crea una ventana, en última instancia con las funciones *CreateWindow* o *CreateWindowEx*, Windows usa la información de la clase, previamente registrada, para saber qué procedimiento de ventana tiene que utilizar.

La función que actúa como procedimiento de ventana recibirá cuatro parámetros: el identificador de la ventana a la que se dirige el mensaje, el mensaje propiamente dicho y dos datos asociados a éste. Como función que es, el procedimiento de ventana deberá devolver un código, cuyo significado dependerá directamente del mensaje que se haya recibido. Por lo tanto, en Visual Basic podríamos implementar una función de proceso de mensajes tal y como se muestra en el siguiente fragmento.

```
 Function ProcesoMensajes(ByVal hWnd As Long, ByVal Mensaje As Long, _
                          ByVal wParam As Long, ByVal lParam As Long) As Long
        ...
End Function
```

Los cuatro parámetros se reciben por valor y son de tipo entero. El identificador de ventana es análogo a la propiedad *hWnd* que tienen la mayoría de los controles Visual Basic, así como los formularios. El mensaje se representa mediante una constante, que podemos recuperar usando el Visor API. Los dos parámetros restantes serán datos asociados al mensaje, su significado dependerá de qué mensaje se reciba.

Subclasificar una ventana

Cuando creamos un programa usando Visual Basic, no tenemos que preocuparnos de registrar clases de ventanas, ni siquiera de crear las propias ventanas, estas tareas las efectúa automáticamente Visual Basic. No podemos, por lo tanto, establecer una función nuestra como procedimiento de ventana en la propia clase de ventana.

Al crearse una ventana Windows asigna espacio en memoria para una estructura de datos privada, estructura que, entre otros elementos, contiene la dirección del procedimiento de esa ventana en particular. Inicialmente la información almacenada en la estructura de datos privada se copia de la estructura de la clase de ventana, pero Windows no impide que se modifique. Disponiendo del identificador de la ventana, que podemos recuperar en cualquier momento de la propiedad *hWnd*, lo único que necesitamos es un medio para acceder a su estructura de datos privada y modificar la dirección del procedimiento actual, estableciendo la de nuestra función.

Mediante la función *SetWindowLong* es posible modificar ciertos apartados de la estructura de datos privada de una ventana, entre ellos la dirección del procedimiento de ventana. Al llamar a esta función deberemos pasar tres parámetros: el identificador de la ventana cuyos datos queremos alterar, una constante indicando qué dato de la estructura se va a modificar y el nuevo valor a establecer. *SetWindowLong* devuelve el antiguo valor que tuviese el dato modificado, lo que nos permite conservarlo para una posterior restitución. En la tabla 8.1 se enumeran las constantes que identifican los diferentes apartados que podemos modificar.

Tabla 8.1: Constantes para la función *SetWindowLong*

Constante	Dato a modificar
GWL_EXSTYLE	Estilo extendido de la ventana
GWL_STYLE	Estilo básico de la ventana
GWL_WNDPROC	Dirección del procedimiento de la ventana
GWL_HINSTANCE	Identificador de la aplicación
GWL_ID	Identificador de la ventana
GWL_USERDATA	Dato de usuario

Por lo tanto, suponiendo que tuviésemos un control y la función **ProcesoMensajes** anterior en un módulo de código estándar, podríamos subclasificar el control con una sentencia como la siguiente.

```
 Private ProcAnterior As Long
...
ProcAnterior = SetWindowLong(hWnd, GWL_WNDPROC, AddressOf ProcesoMensajes)
```

Subclasificada la ventana, en este caso el control, todos los mensajes enviados a él pasarán por nuestra función. Si no se procesa ninguno de estos mensajes el control quedará inservible, ya que no sabrá como dibujarse ni reaccionar a acción alguna. Todos aquellos mensajes que no vayan a ser procesados por nuestro código se deben pasar al anterior

procedimiento de ventana, para lo cual será preciso usar la función *CallWindowProc*. Al llamarla facilitaremos como primer parámetro la dirección del procedimiento, que hemos guardado en la variable **ProcAnterior**, y a continuación los mismos cuatro parámetros que recibimos en **ProcesoMensajes**, en el mismo orden.

El encadenamiento de un procedimiento de ventana con el anterior es fundamental, ya que permite que una misma ventana se subclasifique varias veces sin por ello modificar su funcionamiento. También es de vital importancia restituir la anterior dirección en el momento en que nuestro programa o control se vaya a descargar. Con este fin usaremos de nuevo la función *SetWindowLong*, pasando como tercer parámetro la variable en la cual habíamos almacenado la dirección original.

Un componente para uso de palancas de juegos

Veamos un ejemplo práctico de cómo subclasificar una ventana, en este caso un control, creando un componente que nos facilite el trabajo con palancas de juegos o *joytiscks*. Para utilizar este dispositivo existen una serie de funciones en la API de Windows, mediante las cuales podemos saber cuántas palancas de juegos hay conectadas, obtener información acerca de ellas, etc. Una palanca de juegos de tipo estándar dispone de entre dos y cuatro botones, cada uno de los cuales puede estar pulsado o no. Asimismo, lo habitual es que tengan dos o tres ejes, a los que se suele denominar X, Y y Z.

En lugar de estar llamando periódicamente a una función para saber si se ha pulsado o liberado un botón, o si se ha registrado un movimiento en alguno de los ejes, podemos indicar a Windows que nos envíe un mensaje cada vez que se produzca alguno de esos eventos. Para poder recibir los mensajes necesitaremos subclasificar una ventana, tarea que habría que realizar en cada programa en el que se fuese a utilizar una palanca de juegos.

Desarrollando un control se facilita el trabajo, puesto que toda la complejidad de la subclasificación y gestión de los mensajes se realiza en su interior. El usuario del control no tiene mas que insertarlo en un formulario, establecer propiedades y responder a los eventos, sin más.

Estructura del control

Este control, al que llamaremos **PAXJoystick**, será no visible durante la ejecución, por lo que daremos el valor *True* a la propiedad *InvisibleAtRuntime*. Durante el diseño aparecerá simplemente como un gráfico, que asignaremos a las propiedades *Picture* y *ToolboxBimap*. En la tabla 8.2 se resumen las propiedades y eventos que tendrá.

Tabla 8.2: **Propiedades y eventos del control PAXJoystick**

Miembro	**Comentario**
UsarDispositivo	Propiedad con el identificador del dispositivo a usar, podrá ser **PAXJoy1** o **PAXJoy2**
GenerarEventos	Propiedad de tipo *Boolean* que indicará si hay que generar eventos o no
MovimientoMinimo	Propiedad que especificará el movimiento mínimo que se ha de registrar en el dispositivo antes de producirse un evento
EstaConectado	Propiedad sólo de lectura que contendrá *True* si hay una palanca conectada o *False* en caso contrario
PosX, PosY, PosZ	Propiedades sólo de lectura con las coordenadas en los tres ejes
Botones	Propiedad sólo de lectura con el estado de los botones
OnButtonUp	Evento que se generará al liberarse un botón
OnButtonDown	Evento que se generará al pulsarse un botón
OnMove	Evento que se generará al registrar un movimiento de la palanca

Mediante la propiedad **UsarDispositivo** podremos elegir cual de los dos conectores de *joystick* queremos usar. Tras hacer esta selección podemos consultar el valor de la propiedad **EstaConectado**, comprobando si hay o no una palanca de juegos conectada. En caso afirmativo podemos consultar las propiedades **PosX**, **PosY** y **PosZ** para conocer la posición actual de la palanca en los tres ejes. Estos parámetros suelen estar comprendidos entre 0 y 65535, aunque su tipo es *Long* y, por lo tanto, el rango de valores podría ser mayor. La propiedad **Botones** devuelve un entero con el estado actual de los botones. Para comprobar el estado de uno de ellos deberemos usar el operador lógico *And* junto con una de las constantes de la enumeración **PAXJoyBotones**. Si damos el valor *True* a la propiedad **GenerarEventos**, será el propio control el que genere eventos **OnButtonUp**, **OnButtonDown** y **OnMove** según la acción que se detecte. La mayoría de las palancas de juegos, sobre todo las analógicas, no son capaces de mantener una posición estable, por lo que las coordenadas de los ejes pueden sufrir modificaciones aleatorias y continuas. Para evitar que esto se traduzca en una generación continua de eventos, es posible establecer un movimiento mínimo mediante la propiedad

MovimientoMinimo.

Todo este funcionamiento se basa en el uso de una serie de funciones de la API de Windows, como son *joyGetNumDevs*, *joyGetPos*, *joySetCapture*, *joyReleaseCapture* y *joySetThreshold*.

Para que el control pueda recibir los eventos generados por el dispositivo será preciso hacer una subclasificación, fin con el cual tendremos que añadir un módulo estándar al proyecto. En dicho módulo alojaremos la función que actuará como procedimiento de ventana. El enlace entre el control y el módulo lo efectuaremos mediante una variable pública, que almacenará una referencia al control subclasificado.

Inicialización y destrucción del control

Comencemos por analizar el código encargado de inicializar y destruir el control. Cuando se produzca el evento *Initialize* el control acabará de crearse, siendo el momento apropiado para efectuar la subclasificación. En el método asociado a ese evento enlazaremos el control con el módulo estándar, asignando la referencia *Me* a la variable **ObjetoAsociado** del **Modulo**. Seguidamente usamos la función *SetWindowLong* para subclasificar el control, estableciendo como nuevo procedimiento de ventana la función **ProcesoMensaje** que hay en **Modulo**. La dirección del antiguo procedimiento la conservaremos en la variable pública **ProcedimientoAnterior**, también definida en el módulo estándar.

```
        ' Al inicializar el control
Private Sub UserControl_Initialize()
    Set Modulo.ObjetoAsociado = Me
    Modulo.ProcedimientoAnterior = _
        SetWindowLong(hWnd, GWL_WNDPROC, AddressOf Modulo.ProcesoMensaje)
End Sub
```

Tras el evento *Initialize* será preciso inicializar las propiedades, *InitProperties*, o leerlas, *ReadProperties*, según que el control se acabe de crear o se esté recuperando de un flujo de datos. En el primer caso asignamos a las propiedades **UsarDispositivo**, **GenerarEventos** y **MovimientoMinimo** sus valores por defecto, mientras que en el segundo se recuperan dichas propiedades usando los mismos valores como predeterminados. Observe que en ambos casos la asignación se hace a la propiedad, usando el método de asignación, y no directamente a las variables **PUsarDispositivo**, **PGenerarEventos** y **PMovimientoMinimo**. Esto es así porque la asignación de esos valores ha de traducirse en las correspondientes llamadas a las funciones de la API de Windows.

```
        ' Inicialización de propiedades
Private Sub UserControl_InitProperties()
    UsarDispositivo = PAXJoy1 ' Primer dispositivo
    GenerarEventos = True ' generando eventos
    MovimientoMinimo = 128 ' con un movimiento mínimo de 128
End Sub

    ' Lectura de las propiedades
Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    With PropBag
        UsarDispositivo = .ReadProperty("UsarDispositivo", PAXJoy1)
        GenerarEventos = .ReadProperty("GenerarEventos", True)
        MovimientoMinimo = .ReadProperty("MovimientoMinimo", 128)
    End With
End Sub
```

Cuando el control va a ser destruido es preciso guardar los valores de las propiedades, respondiendo adecuadamente al evento *WriteProperties*, y eliminar la subclasificación de la ventana, tarea ésta que llevaremos a cabo en el método asociado al evento *Terminate*. Usando la misma función *SetWindowLong* restituiremos la antigua dirección, tras lo cual asignamos el valor *Nothing* a la variable del módulo que mantenía la referencia al control, asegurándonos así su correcta destrucción. De no efectuar esta asignación el control no sería destruido, por lo que a pesar de que el programa terminase el proceso de liberación no se podría completar satisfactoriamente.

```
 ' Al destruir el control
Private Sub UserControl_Terminate()
        ' restablecemos su procedimiento original
    SetWindowLong hWnd, GWL_WNDPROC, Modulo.ProcedimientoAnterior
    Set Modulo.ObjetoAsociado = Nothing ' y destruimos la referencia
End Sub
```

Selección del dispositivo a utilizar

Creado el control lo normal es que el siguiente paso sea la selección del dispositivo que se va a usar, para lo cual se asignará el valor adecuado a la propiedad **UsarDispositivo**. En el método de asignación de esta propiedad lo primero que tenemos que hacer es comprobar si el dispositivo que se estaba utilizando hasta el momento estaba "capturado", caso en el cual hay que "liberarlo". Tras esto ya se puede fijar el nuevo dispositivo, que dejaremos en el mismo estado en que se encontraba el anterior.

Para que una palanca de juegos nos envíe mensajes cada vez que se produzca una modificación en el estado de los botones o la posición, es necesario llamar previamente a la función *joySetCapture*. Esta función necesita cuatro parámetros: el identificador de la ventana a la que se han de enviar los mensajes, el identificador del dispositivo a capturar, un intervalo de tiempo que se usará como periodo de envío de los mensajes y un indicador. Tras llamar a esta función se dice que hemos "capturado" el dispositivo, ya que en cierta forma tomamos en exclusiva su funcionamiento. De hecho, si un programa hace una segunda llamada a *joySetCapture* obtendrá un error como resultado. Por eso es necesario liberar el dispositivo en cuanto no lo estemos usando, simplemente llamando a la función *joyReleaseCapture* pasando como parámetro el identificador de la palanca de juegos.

El tercer parámetro de *joySetCapture* especifica un tiempo que ha de transcurrir entre el envío de dos mensajes consecutivos a la ventana. Si pasamos el valor 0 y como cuarto parámetro el valor *True*, estaremos comunicando a Windows que no genere un mensaje periódico, sino sólo cuando se haya pulsado o liberado un botón o bien cuando se registre un movimiento superior a una cierta magnitud. Esta magnitud se establece mediante la función *joySetThreshold*, que necesita como primer parámetro el identificador del dispositivo y como segundo el factor citado.

El código correspondiente al método de asignación a la propiedad **UsarDispositivo**, que puede ver a continuación, efectúa todo este proceso sólo si el valor que se va a asignar es diferente al que existe en ese momento, es decir, si efectivamente se va a cambiar el dispositivo, ya que de lo contrario no tiene sentido realizar el trabajo.

```
Public Property Let UsarDispositivo(NuevoValor As PAXJoyDispositivo)
   If PUsarDispositivo <> NuevoValor Then ' si se modifica realmente
       If PGenerarEventos And Ambient.UserMode Then ' si se tiene el dispositivo
           joyReleaseCapture PUsarDispositivo
       End If
       PUsarDispositivo = NuevoValor ' asignamos el nuevo dispositivo
       If PGenerarEventos And Ambient.UserMode Then ' si se tenía el dispositivo
           joySetCapture hWnd, PUsarDispositivo, 0, True ' dejarlo como estaba
           joySetThreshold PUsarDispositivo, PMovimientoMinimo
       End If
   End If
End Property
```

Observe que las llamadas a *joySetCapture*, *joySetThreshold* y *joyReleaseCapture* se realizan sólo en caso de que el modo actual sea de ejecución, ya que no tiene sentido llamarlas cuando el control se encuentra en modo de diseño puesto que no es posible recuperar información alguna.

*Las propiedades **GenerarEventos** y **MovimientoMinimo***

Los métodos de lectura de estas propiedades, al igual que el de la propiedad **UsarDispositivo**, se limitan a devolver el valor almacenado en la variable privada correspondiente, sin efectuar ninguna operación adicional.

Al modificarse la propiedad **GenerarEventos** habrá que capturar el dispositivo, si el nuevo valor es *True*, o liberarlo, en caso de que sea *False*. Ambas operaciones se efectúan sólo si estamos en modo de ejecución.

```
Public Property Let GenerarEventos(NuevoValor As Boolean)
   PGenerarEventos = NuevoValor ' guardamos el valor
   If Ambient.UserMode Then ' si estamos en ejecución
       If PGenerarEventos Then ' si hay que generar eventos
           joySetCapture hWnd, PUsarDispositivo, 0, True ' capturar el dispositi
       Else ' en caso contrario
           joyReleaseCapture PUsarDispositivo ' liberarlo
       End If
   End If
End Property
```

De forma análoga, la asignación de un cierto valor a la propiedad **MovimientoMinimo** ha de traducirse en una llamada a la función *joySetThreshold*, estableciéndose el nuevo movimiento mínimo que ha de registrarse antes de que se envíe un mensaje de notificación.

```
 Public Property Let MovimientoMinimo(NuevoValor As Long)
    PMovimientoMinimo = NuevoValor ' guardamos el valor
        ' si estamos ejecutando
    If Ambient.UserMode Then _
        joySetThreshold PUsarDispositivo, NuevoValor ' lo pasamos al dispositivo
End Property
```

Propiedades informativas

Seleccionado el dispositivo que se va a utilizar, lo normal será comprobar si en realidad hay o no conectada una palanca de juegos. Con este fin se consultará el valor de la propiedad sólo de lectura **EstaConectado**. Dicha propiedad ha de devolver *True* o *False*, según haya o no un *joystick* conectado en ese momento. Tendremos que usar dos nuevas funciones para determinar si un dispositivo está o no disponible y conectado: *joyGetNumDevs* y *joyGetPos*. La primera de ellas devuelve el número de palancas de juegos que es posible conectar de forma simultánea, por regla general dos. La segunda recupera el estado actual de los botones y posición de la palanca, devolviendo el valor *JOYERR_NOERROR* si no hay ningún problema.

El número de dispositivo seleccionado, que se almacena en la variable **PUsarDispositivo**, siempre habrá de estar comprendido entre 0 y el número devuelto por *joyGetNumDevs* menos 1. Es decir, el valor que retorna *joyGetNumDevs* debe ser mayor que **PUsarDispositivo**, de lo contrario el dispositivo elegido no será válido.

joyGetNumDevs devuelve el número de dispositivos que es posible conectar, no el que hay conectado en este momento. Si tenemos instalado el controlador estándar de palancas de juego de Windows, esta función devolverá el valor 2 a pesar de que no tengamos ningún dispositivo conectado. Para comprobar si hay o no conectada una palanca de juegos es preciso consultarla, preguntándole el estado de los botones y la posición. Para ello se usa la función *joyGetPos*, pasando como primer parámetro el identificador del dispositivo a consultar y como segundo una variable del tipo *JOYINFO*. Si la llamada a *joyGetPos* devuelve el valor *JOYERR_NOERROR* es que la palanca está conectada.

```
      ' Propiedad sólo de lectura que devuelve True si hay un joystick conectado
Public Property Get EstaConectado() As Boolean
    Dim Inf As JOYINFO

        ' Devolvemos True si joyGetNumDevs devuelve un número mayor
        ' que el del dispositivo que queremos usar y, además, la
        ' llamada a joyGetPos no devuelve error alguno
    EstaConectado = joyGetNumDevs > PUsarDispositivo And _
                    joyGetPos(PUsarDispositivo, Inf) = JOYERR_NOERROR
End Property
```

El resto de las propiedades informativas, todas de sólo lectura, se limitan a recuperar y devolver uno de los datos de la estructura *JOYINFO*. Este tipo cuenta con los miembros enumerados en la tabla 8.3, todos ellos de tipo *Long*. En lugar de llamar a la función *joyGetPos* en el método de lectura de las propiedades **PosX**, **PosY**, **PosZ** y **Botones**, codificaremos un método privado que será el encargado de realizar esa llamada, devolviendo como resultado la estructura de datos *JOYINFO*. De esta forma los métodos de lectura de las propiedades se pueden limitar a devolver el miembro que corresponda, sin más.

Tabla 8.3: **Miembros del tipo *JOYINFO***

Nombre	**Contenido**
wXpos	Posición en el eje X
wYpos	Posición en el eje Y
wZpos	Posición en el eje Z
wButtons	Estado de los botones

```
     ' Método privado para consultar el estado del dispositivo
Private Function Estado() As JOYINFO
    Dim Inf As JOYINFO
```

```
    joyGetPos PUsarDispositivo, Inf ' consultamos el dispositivo
    Estado = Inf ' y devolvemos la información
End Function

    ' Propiedades sólo de lectura para recuperar las coordenadas
    ' de la palanca de juegos
Public Property Get PosX() As Long
    PosX = Estado.wXpos
End Property

Public Property Get PosY() As Long
    PosY = Estado.wYpos
End Property

Public Property Get PosZ() As Long
    PosZ = Estado.wZpos
End Property

    ' Propiedad sólo de lectura para recuperar el estado de los botones
Public Property Get Botones() As Long
    Botones = Estado.wButtons
End Property
```

Proceso de los mensajes y generación de eventos

Suponiendo que la propiedad **GenerarEventos** del control esté activada, todos los mensajes que Windows envíe a nuestro control pasarán previamente por la función **ProcesaMensaje**, que hemos alojado en un módulo de código estándar. Esta función comprobará si el mensaje que se recibe, en el segundo parámetro, es uno de los relacionados con las palancas de juegos, en cuyo caso lo pasará al método **Evento** del control. Indistintamente de si se da este paso o no, el mensaje siempre será enviado al anterior procedimiento de ventana mediante la función *CallWindowProc*.

```
     ' Función que se encargará de procesar los mensajes
Function ProcesoMensaje(ByVal hWnd As Long, ByVal Mensaje As Long, _
                        ByVal wParam As Long, ByVal lParam As Long) As Long

        ' Si es uno de los mensajes que nos interesa
    If (Mensaje >= MM_JOY1BUTTONDOWN And Mensaje <= MM_JOY2BUTTONUP) Or _
       (Mensaje >= MM_JOY1MOVE And Mensaje <= MM_JOY2ZMOVE) Then
        ObjetoAsociado.Evento Mensaje ' lo notificamos a nuestro cliente
    End If

        ' En cualquier caso llamar a CallWindowProc y devolver
        ' el valor que obtengamos
    ProcesoMensaje = CallWindowProc(ProcedimientoAnterior, hWnd, Mensaje, wParam,
End Function
```

El método **Evento** del control **PAXJoystick** decidirá qué evento generar según el mensaje que se haya recibido. A pesar de ser público, este método está pensado para utilizarse solamente como medio de comunicación entre el módulo estándar y el control, por lo que usando la ventana de atributos de procedimiento lo haremos oculto.

```
     ' Este procedimiento será llamado desde el módulo
    ' de código para notificar un evento
Public Sub Evento(Mensaje As Long)
    Select Case Mensaje
        Case MM_JOY1BUTTONDOWN, MM_JOY2BUTTONDOWN
            RaiseEvent OnButtonDown
        Case MM_JOY1BUTTONUP, MM_JOY2BUTTONUP
            RaiseEvent OnButtonUp
        Case Else
            RaiseEvent OnMove
    End Select
End Sub
```

Con esto ya tenemos terminado nuestro control, el primero que hemos subclasificado. Aunque funcionará correctamente, lo vamos a ver en un momento, el método de comunicación usado entre el control y el módulo estándar es muy débil.

*Uso de un **PAXJoystick***

Utilizar un control subclasificado en el interior del entorno de desarrollo de Visual Basic es bastante peligroso. Cualquier error que nos haga entrar en el modo de interrupción puede provocar el bloqueo completo del entorno, asegúrese de guardar siempre antes su trabajo. Una alternativa más segura consiste en compilar siempre el código y ejecutarlo desde fuera del entorno de desarrollo.

Para comprobar el funcionamiento de nuestro control vamos a diseñar un formulario como el que se muestra en la figura 8.5. Tenemos cuatro controles **TextBox** para mostrar el estado de los botones, otros tres para mostrar la posición en cada uno de los ejes, un botón para activar/desactivar la generación de eventos, otro para actualizar la información que hay en la ventana y, por último, un control **PAXJoystick**.

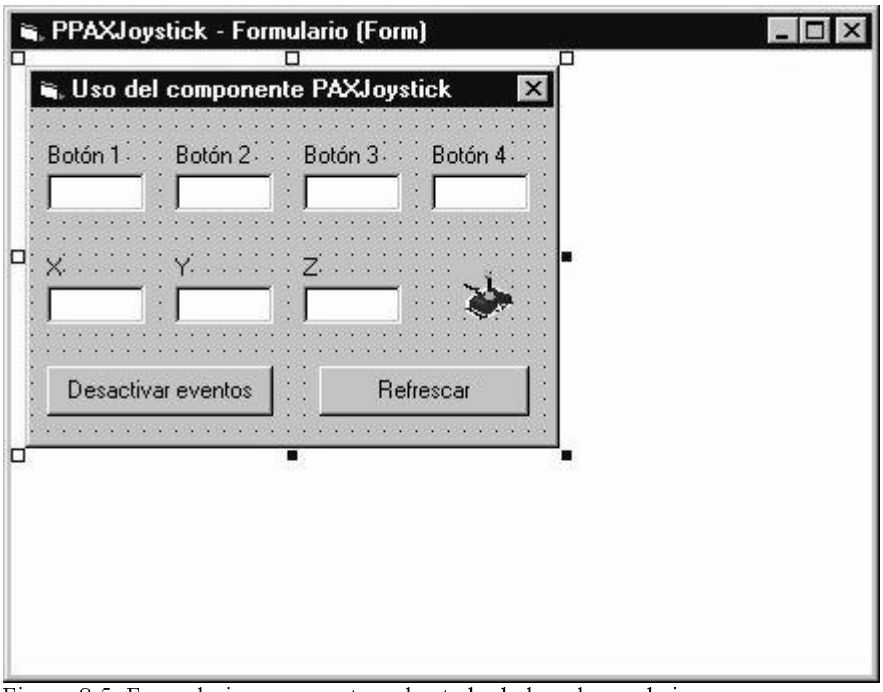

Figura 8.5: Formulario para mostrar el estado de la palanca de juegos

Al cargarse el formulario se consultará el valor de la propiedad **EstaConectado**, notificando si hay o no conectada una palanca de juegos. En caso negativo el programa termina directamente. Si tenemos un *joystick* conectado a la salida elegida en la propiedad **UsarDispositivo**, cada vez que se pulse el botón **Refrescar** recuperaremos los datos de las propiedades informativas. El estado de los botones lo indicaremos cambiando el color de fondo de los **TextBox**, que será rojo si el botón correspondiente está pulsado o blanco en caso contrario. Las posiciones en los distintos ejes se mostrarán directamente, asignando el valor de las propiedades **PosX**, **PosY** y **PosZ** a la propiedad *Text* de las cajas de texto.

La pulsación del botón que hay a la izquierda activará/desactivará la generación de eventos por parte del control, para lo cual invertirá el estado de la propiedad **GenerarEventos**. Según el estado actual, el botón mostrará un título u otro. En los métodos asociados a los eventos **OnButtonDown**, **OnButtonUp** y **OnMove** simplemente llamaremos al método correspondiente al evento *Click* del botón **Refrescar**, actualizando la información mostrada.

Como puede ver en el siguiente fragmento, para comprobar el estado de los botones no se han usado las constantes que los representan. En su lugar se ha utilizado un bucle cuyo contador sirve como índice de una operación de potenciación cuya base es 2. El resultado es exactamente el mismo, puesto que los valores obtenidos (1, 2, 4 y 8) son los que representan esas constantes, sin embargo el código es más corto.

```
    ' Cada vez que se pulse el botón Refrescar
Private Sub Command1_Click()
    Dim Contador As Integer

    With PAXJoystick1 ' usamos el control PAXJoystick
        For Contador = 0 To 3 ' para recuperar el estado de los cuatro botones
```

```
            tbBoton(Contador).BackColor = _
                IIf((.Botones And 2 ^ Contador) <> 0, vbRed, vbWhite)
        Next
        tbX = .PosX ' y las posiciones
        tbY = .PosY
        tbZ = .PosZ
    End With
End Sub

    ' La pulsación del botón Activar/Desactivar eventos
Private Sub Command2_Click()
    With PAXJoystick1
            ' invertimos el estado de la propiedad GenerarEventos
        .GenerarEventos = Not .GenerarEventos
            ' según el estado actual fijamos el título del botón
        Command2.Caption = IIf(.GenerarEventos, "Desactivar eventos", "Activar ev
    End With
End Sub

    ' Al cargar el formulario
Private Sub Form_Load()
        ' Si hay una palanca de juegos
    If PAXJoystick1.EstaConectado Then
            ' lo notificamos
        MsgBox "Hay una palanca de juegos conectada"
    Else ' en caso contrario avisamos
        MsgBox "No hay una palanca de juegos conectada"
        Unload Me ' y descargamos
    End If
End Sub

    ' Cada vez que se produzca un evento
Private Sub PAXJoystick1_OnButtonDown()
    Command1_Click
End Sub
```

Si ejecuta el programa y no dispone de una palanca de juegos conectada a su sistema no podrá ver resultado alguno. En la figura 8.6 puede ver el formulario mostrando el estado actual de los botones y la posición en los ejes X e Y.

Figura 8.6: El programa PPAXJoystick en funcionamiento

La importancia de la encapsulación

El diseño del control que acabamos de crear cuenta con el mismo problema que teníamos inicialmente con el control **PAXEnumeraVentanas**, la falta de encapsulación del código alojado en el módulo estándar. Este problema puede ser el origen de diversos problemas que nos puede llevar mucho tiempo descubrir, sobre todo si no somos conscientes de que las variables globales de un componente ActiveX son compartidas por las diferentes copias del objeto.

Supongamos que inserta en un formulario un control **PAXJoystick**. Justo en ese momento se produce la inicialización, en la cual se asigna la dirección del anterior procedimiento de ventana a la variable **ProcedimientoAnterior**, además de asignarse a **ObjetoAsociado** una referencia al propio control. Estas dos variables son globales, lo cual quiere decir que

son compartidas. Si ahora inserta en el formulario un segundo control **PAXJoystick**, de nuevo se asigna a **ProcedimientoAnterior** la antigua dirección del procedimiento de ventana, perdiéndose el anterior contenido de esta variable. La variable **ObjetoAsociado**, por su parte, ahora apuntará al segundo control.

En este punto nos encontramos con que la función que hay en el módulo estándar está recibiendo mensajes de los dos controles insertados, sin embargo en la llamada a *CallWindowProc* todos los mensajes se pasan al procedimiento cuya dirección se almacena en **ProcedimientoAnterior**, es decir, al procedimiento de ventana del segundo control insertado. De igual forma, cuando se encuentra un mensaje relativo a la palanca de juegos se notifica al control cuya referencia se almacena en **ObjetoAsociado**, que también es el segundo. El primer control deja de funcionar por completo.

La situación descrita, sin embargo, puede no ser la peor. Partiendo del escenario anterior imagine que eliminamos el segundo de los controles. En ese momento se restituye la antigua dirección del procedimiento de ventana del control, tras lo cual se asigna el valor *Nothing* a la variable **ObjetoAsociado**. En este momento la función **ProcesaMensaje** aún puede recibir mensajes del primer control, puesto que éste sigue subclasificado. Dichos mensajes serán pasados mediante *CallWindowProc* a la función cuya dirección se encuentra en **ProcedimientoAnterior**, es decir, a la dirección del procedimiento de ventana original del segundo control, que en ese momento ya no existe. Si el mensaje recibido notifica un evento de la palanca de juegos, se intentará pasar al control asociado usando la referencia almacenada en **ObjetoAsociado**, pero en este momento dicha variable contiene el valor *Nothing*. Como puede imaginarse, el resultado es desastroso.

La solución a estos problemas es clara: necesitamos encapsular las variables públicas del módulo, de tal forma que cada control cuente con las suyas privadas. La forma de conseguir esto, sin embargo, no está tan clara. Al diseñar el control **PAXEnumeraVentanas** encontramos una manera de hacerlo bastante eficiente, pero que no es válida en este caso. La función **ProcesaMensaje**, que es a la que Windows llama directamente, no cuenta con un parámetro adicional facilitado por nosotros, como sí ocurría con la función a la que llamaba *EnumWindows*.

Datos de usuario en una ventana

Anteriormente describíamos una ventana como una estructura de datos, en la cual existen miembros que establecen el estilo de la ventana, el procedimiento de gestión de mensajes y otros elementos. En dicha estructura existe un espacio reservado para datos de usuario, espacio que está totalmente a nuestra disposición.

Para guardar un valor en ese espacio usaremos la función *SetWindowLong* que ya conocemos, facilitando como segundo parámetro la constante *GWL_USERDATA*. Con el fin de recuperar el dato, cuando nos interese, usaremos la función *GetWindowLong*, que tomando como parámetros el identificador de ventana y la citada constante devuelve el valor almacenado.

El espacio reservado para datos de usuario en la estructura de una ventana es de sólo cuatro bytes, es decir, un valor de tipo *Long*. Podríamos usar este espacio para almacenar, por ejemplo, la dirección del antiguo procedimiento de ventana que hasta ahora guardábamos en la variable **ProcedimientoAnterior**. Sin embargo seguiríamos teniendo el problema de la referencia al control, conservada en la variable **ObjetoAsociado**. En realidad tenemos dos problemas: necesitamos almacenar más datos de los que podemos, por un parte, y queremos guardar una referencia a un objeto en un espacio pensado para un *Long*.

La solución a estos problemas la encontraremos en el uso de punteros, tal y como vamos a ver en el punto siguiente. No se trata de la técnica más fácil pero seguramente sí de la más efectiva.

Punteros a variables, cadenas y objetos

¿Es necesario el uso de punteros para programar? Si hace esta pregunta a un grupo de programadores seguramente obtendrá multitud de respuestas, pero la mayoría de ellas se podrán agrupar en dos categorías: "El uso de punteros es fundamental para la programación" y "Hay que evitar por todos los medios el uso de punteros, es una aberración" La primera respuesta la obtendrá de aquellos programadores que han utilizado punteros siempre, léase aquellos que desde sus inicios han usado lenguajes como C/C++. La segunda es propia de "puristas" que consideran que trabajar con direcciones de memoria en un programa es algo que debería estar prohibido, por los riesgos que conlleva, los problemas de portabilidad, etc.

¿Qué respuesta darían los programadores Visual Basic a esa pregunta? Seguramente una contestación resignada: "En Visual Basic no es posible trabajar con punteros, una lástima"; o bien una respuesta de satisfacción: "Por suerte en Visual Basic no existen los punteros" Desde luego Visual Basic no es un lenguaje diseñado para trabajar de forma cómoda con punteros pero, por suerte, no es algo imposible. Si piensa que, en ocasiones, el uso de los punteros puede estar justificado para conseguir una cierto fin está de suerte, porque en Visual Basic 5 es posible trabajar con punteros.

Existen tres funciones de las cuales no encontrará documentación en los manuales de Visual Basic, ni en la ayuda en línea. Ni siquiera podrá encontrarlas usando el Examinador de objetos, puesto que están contenidas en un módulo oculto. Sí puede encontrarlas, sin embargo, usando la herramienta OLEVIEW que utilizamos en un capítulo anterior para inspeccionar el contenido de las librerías de tipos. Cargando la librería de tipos del archivo VBA5.DLL, que encontrará en el propio directorio de Visual Basic, podrá ver un módulo llamado **_HiddenModule**. Las funciones que nos interesan son las últimas tres de este módulo, destacadas en la imagen de la figura 8.7. Sus nombres son *VarPtr*, *StrPtr* y *ObjPtr*.

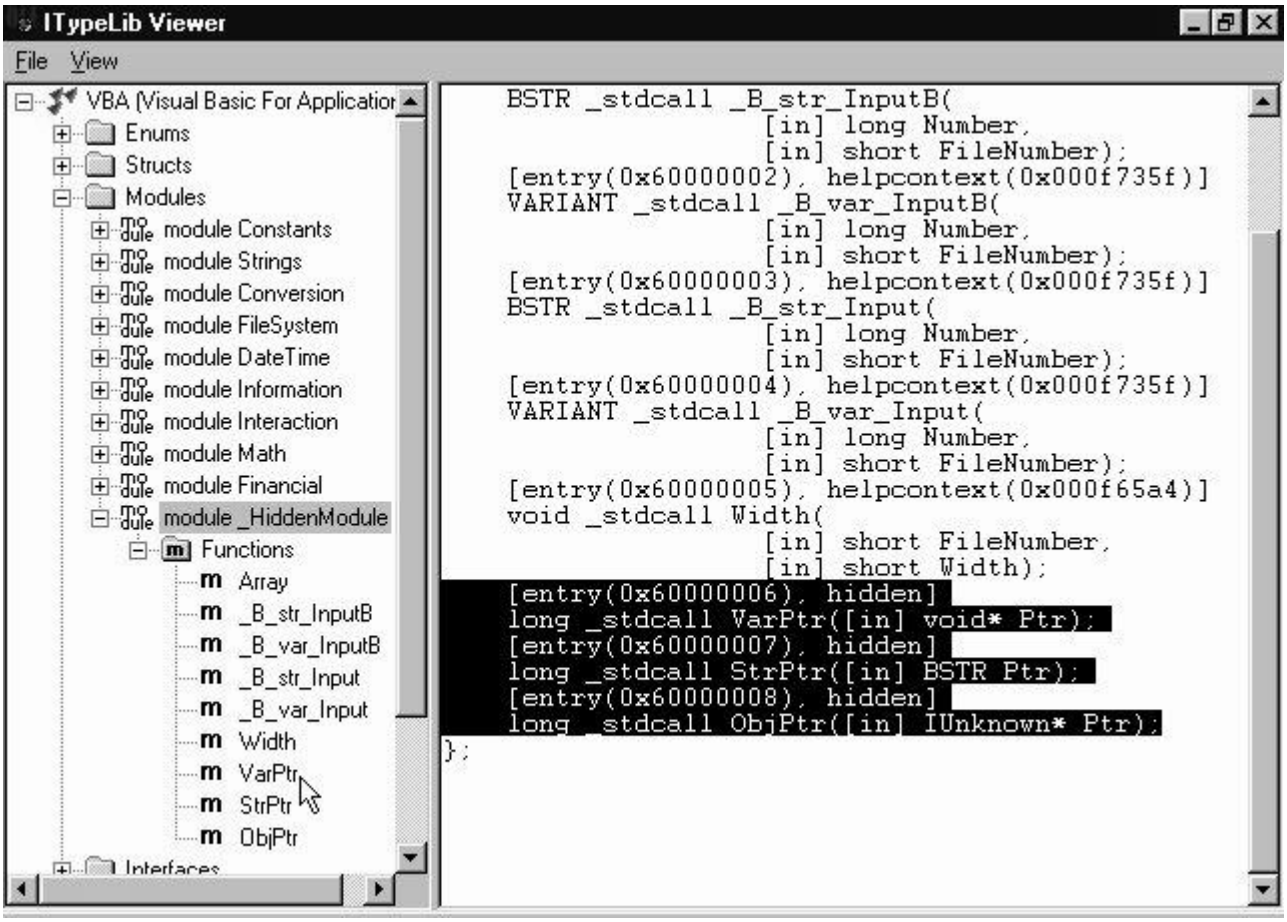

Figura 8.7: Contenido del módulo oculto existente en la librería VBA5.DLL

Si escribe en el Editor de código el nombre de cualquiera de estas tres funciones y pulsa un espacio, verá que aparece una indicación de los parámetros que es necesario pasar. Esto nos demuestra que las funciones son reconocidas por Visual Basic, de lo contrario se las trataría como a cualquier otro identificador desconocido.

La función *VarPtr*

Esta función no le resultará nueva si lleva muchos años programando con BASIC. *VarPtr*, usada conjuntamente con *Peek* y *Poke*, permitía a los programadores de entonces efectuar operaciones de bajo nivel, sin las cuales no era posible conseguir muchos resultados. Se podía, por ejemplo, declarar una matriz de bytes, escribir en ella con *Poke* y después ejecutarla como un programa en código máquina obteniendo la dirección de la matriz con *VarPtr*.

La función *VarPtr* acepta como parámetro una variable de cualquier tipo, devolviendo un entero *Long* con la dirección de memoria en la que se almacena el contenido de dicha variable. Esta dirección puede ser almacenada en otra variable, pasada como parámetro a una función o usada en cualquier otro contexto en la que se puede utilizar un *Long*.

¿Cuándo necesitará usar la función *VarPtr*? En caso de que haya que pasar a una función de la API de Windows la dirección de una variable, siempre puede declarar el parámetro correspondiente como *ByRef*. Algunas funciones, sin embargo, necesitan como parámetro una estructura de datos en cuyo interior existe punteros a variables u otras estructuras. En casos así *VarPtr* facilitará el trabajo, ya que bastará con realizar una asignación como la siguiente:

```
Estructura.Puntero = VarPtr(Variable)
```

Éste es uno de los pocos casos en que encontraremos utilidad a *VarPtr*. La manipulación del contenido de las variables, a partir de la dirección facilitada, siempre quedará en manos de la función de la API a la que llamemos. Aunque hay pocos

casos que lo justifiquen, también desde Visual Basic es posible alterar el contenido de esa zona de memoria, como veremos en un punto posterior.

La función *StrPtr*

El almacenamiento de las cadenas en Visual Basic no es tan sencillo como cabría esperar, a diferencia de lo que ocurre en otros lenguajes en los que una cadena es, simplemente, una secuencia de bytes consecutivos. Cuando declaramos una variable de tipo *String* lo que se aloja en ella es un puntero, inicialmente nulo. Al asignarle un valor, Visual Basic asigna un bloque de memoria y copia en él la cadena, guardando la dirección de dicho bloque en la variable *String*. El bloque que contiene la cadena también almacena, al inicio, un entero *Long* que indica la longitud, así como un carácter nulo al final.

La función *VarPtr* devuelve la dirección de una variable, lo que nos permite saber la posición de memoria en que está almacenado el valor que contiene. El contenido de una variable *String* es a su vez un puntero, por lo que para obtener la dirección real de la secuencia de caracteres que forman la cadena deberemos usar la función *StrPtr*. Puede comprobar la diferencia existente entre los valores devueltos por *VarPtr* y *StrPtr* escribiendo el código siguiente:

```
 Dim Cadena1 As String
Dim Cadena2 As String
Cadena1 = "Hola"
Print "Cadena1 -> ", VarPtr(Cadena1), StrPtr(Cadena1)
Print "Cadena2 -> ", VarPtr(Cadena2), StrPtr(Cadena2)
```

En la figura 8.8 puede ver que los valores devueltos por *VarPtr* son dos direcciones consecutivas, que en su sistema pueden ser, de hecho es lo normal, otras totalmente diferentes. La primera cadena contiene actualmente un valor, por lo que *StrPtr* devuelve la dirección en que se encuentra la secuencia de caracteres. La segunda, por el contrario, está vacía, de ahí que la dirección devuelta por *StrPtr* sea nula.

Figura 8.8: Direcciones de cadenas obtenidas con *VarPtr* y *StrPtr*

Internamente Visual Basic almacena las cadenas en formato Unicode, esto es, usa 16 bits para cada carácter, no 8. Desde la versión 4.0 Visual Basic está basado principalmente en COM, todos sus objetos y controles son objetos COM. La especificación COM establece que todas las cadenas habrán de ser Unicode, no existiendo la posibilidad de trabajar con cadenas ANSI. Visual Basic se encarga de realizar la conversión siempre que es necesario, pero al acceder directamente a la cadena esta conversión no se efectúa, hemos de tenerlo en cuenta.

La función *ObjPtr*

Desde el mismo momento en que comenzamos a trabajar con Visual Basic estamos usando objetos, como el formulario, los controles, las colecciones y nuestros propios componentes. Sabemos que estos elementos son objetos COM, de los cuales podemos tener referencias en variables del tipo correspondiente. Lo que se almacena en esas variables, la propia referencia, no es mas que una dirección, un apuntador a la posición de memoria en la que se encuentra el objeto propiamente dicho, sus interfaces y su código.

Al igual que ocurre con las cadenas, la función *VarPtr* devuelve la dirección de la variable que contiene la referencia, pero no la dirección del objeto en sí. Para conseguir este dato deberemos usar la función *ObjPtr*. Puede hacer una prueba usando el siguiente código, en el que se asignan diferentes referencias de un mismo objeto a varias variables.

```
 Dim ObjetoOriginal As New Collection
Dim ObjetoCopia As Collection
Dim ObjetoGenerico As Object

Print "ObjetoOriginal -> ", VarPtr(ObjetoOriginal), ObjPtr(ObjetoOriginal)
Print "ObjetoCopia    -> ", VarPtr(ObjetoCopia), ObjPtr(ObjetoCopia)
Print "ObjetoGenerico -> ", VarPtr(ObjetoGenerico), ObjPtr(ObjetoGenerico)
Set ObjetoCopia = ObjetoOriginal
Set ObjetoGenerico = ObjetoOriginal
Print "ObjetoCopia/Set-> ", VarPtr(ObjetoCopia), ObjPtr(ObjetoCopia)
```

```
    Print "ObjetoGenerico/Set -> ", VarPtr(ObjetoGenerico), ObjPtr(ObjetoGenerico
```

Lógicamente cada variable es independiente de las demás, por lo cual tiene su propia dirección. Esta la obtenemos con la función *VarPtr*. Inicialmente **ObjetoCopia** y **ObjetoGenerico** no contienen referencia alguna, por ello *ObjPtr* devuelve el valor 0. Tras una asignación podemos comprobar que las tres referencias contienen la dirección de un mismo objeto.

Figura 8.9: Direcciones de objetos devueltas por *VarPtr* y *ObjPtr*

Usando la función *ObjPtr* podemos guardar la dirección de un objeto en la estructura de datos de una ventana, usando la ya conocida *SetWindowLong*. Con esto solucionamos uno de los problemas indicados anteriormente, la imposibilidad de guardar una referencia a un objeto. Seguimos teniendo, no obstante, un problema adicional: sólo podemos guardar un dato de 4 bytes asociado a la ventana.

Manipulación directa de la memoria

Las tres funciones que hemos conocido en los puntos anteriores nos permiten obtener la dirección de memoria en la que se encuentra el contenido de cualquier variable, indistintamente de que sea de un tipo numérico, una cadena o un objeto. Con esto sólo podemos recuperar un puntero y usarlo como parámetro, bien sea para pasarlo a una función de la API de Windows o para guardarlo en una variable. No podemos, sin embargo, manipular directamente el contenido de ese espacio de memoria. No, en Visual Basic no existen las anteriormente mencionadas *Peek* y *Poke*, con las cuales era posible leer o escribir un byte en memoria.

Buscando en las diferentes librerías de Windows, sin embargo, podemos encontrar algunas funciones útiles, entre ellas una llamada *RtlMoveMemory*. Este procedimiento facilita la copia de una zona de memoria a otra, para lo cual es necesario facilitar tres parámetros: dirección de destino, dirección de origen y número de bytes a copiar. Podemos usar esta función en Visual Basic declarándola de la siguiente forma:

```
Private Declare Sub RtlMoveMemory Lib "kernel32" _
   (ByVal DestAddr As Long, ByVal SrcAddr As Long, ByVal Bytes As Long)
```

Los valores devueltos por *VarPtr*, *StrPtr* y *ObjPtr*, punteros en los tres casos, no son peligrosos por sí solos, pero usados conjuntamente con *RtlMoveMemory* pueden ser muy útiles al tiempo que desastrosos. Un fallo insignificante puede causar el bloqueo del programa o el sistema, así de sencillo. En el siguiente código, por ejemplo, se copia el contenido de una variable de tipo entero a otra del mismo tipo. ¿Qué ocurriría si la longitud, el número de bytes a copiar, fuese erróneo? Se escribiría en zonas de memoria que nada tienen que ver con la variable, algo realmente peligroso. En este caso podrá ver que tras usar *RtlMoveMemory* ambas variables contienen el mismo valor, es como si se hubiese efectuado una asignación.

```
    Dim Numero1 As Integer, Numero2 As Integer
   Numero1 = 1234
   Print "Antes -----------------"
   Print "Numero1 -> " & Numero1
   Print "Numero2 -> " & Numero2
   RtlMoveMemory VarPtr(Numero2), VarPtr(Numero1), Len(Numero1)
   Print "Después ---------------"
   Print "Numero1 -> " & Numero1
   Print "Numero2 -> " & Numero2
```

Está claro que para asignar una variable de tipo entero a otra no es necesario usar *RtlMoveMemory*, esto es sólo un ejemplo. En la práctica, no obstante, nos podemos encontrar con casos en los que no existe otra alternativa. Suponga que al subclasificar un control quiere almacenar en la estructura de datos de ventana un puntero al propio componente. No tiene mas que recuperar dicho puntero, con *ObjPtr*, y después copiarlo con *SetWindowLong*. En el procedimiento de ventana, que gestionará los mensajes, podría recuperar la dirección del control con *GetWindowLong* pero, ¿cómo lo puede usar? Lo que obtiene es un puntero, un valor de tipo *Long*, no una referencia a un objeto. Una asignación directa del puntero a una variable que espera una referencia no es posible, pero sí se puede conseguir usando *RtlMoveMemory*.

Un mejor control PAXJoystick

Veamos cómo podemos aprovechar lo que hemos aprendido para mejorar el diseño de nuestro control **PAXJoystick**, encapsulando el funcionamiento del módulo estándar para evitar problemas cuando se usan varios controles de forma simultánea.

Puesto que necesitamos almacenar dos datos: la referencia al objeto que es el control y la anterior dirección del procedimiento de ventana, no podremos alojarlos directamente en la estructura de datos de la ventana, ya que en ella sólo hay espacio para 32 bits. Sí podemos, sin embargo, almacenar los dos datos en una estructura y guardar la dirección de ésta con *SetWindowLong*. Esta es la idea que vamos a desarrollar.

Almacenamiento de la información de enlace

Comenzaremos por definir la estructura de datos que vamos a utilizar, un nuevo tipo al que llamaremos **PAXEnlace**. Puesto que va a ser necesario usarlo tanto en el control como en el módulo estándar, incluiremos la definición en este último. Este nuevo tipo tendrá dos miembros de tipo *Long*: **ControlAsociado** y **ProcedimientoAnterior**. El primero tendrá la dirección del control **PAXJoystick2** asociado, mientras que el segundo almacenará la dirección del anterior procedimiento de ventana.

En el módulo de código del control incluiremos una nueva variable privada, llamada **Enlace**, de tipo **PAXEnlace**. Esta variable no va a ser accesible de ninguna forma para el usuario final del control, nos servirá internamente para guardar la información de enlace entre el módulo estándar y el control mismo. Al producirse el evento de inicialización, momento en el que subclasificaremos, guardaremos en la estructura de la ventana la dirección de la variable **Enlace**, que como puede ver obtenemos mediante la función *VarPtr*. Tras esto recuperamos la dirección del control, con la función *ObjPtr*, y la guardamos en el miembro **ControlAsociado**. Por último modificamos el procedimiento de ventana, guardando la dirección del anterior en **ProcedimientoAnterior**.

```
        ' Al inicializar el control
Private Sub UserControl_Initialize()
        ' guardamos como dato de usuario la dirección
        ' de la variable Enlace
    SetWindowLong hwnd, GWL_USERDATA, VarPtr(Enlace)
    Enlace.ControlAsociado = ObjPtr(Me)
    Enlace.ProcedimientoAnterior = _
        SetWindowLong(hwnd, GWL_WNDPROC, AddressOf Modulo.ProcesoMensaje)
End Sub
```

Observe que lo que guardamos en la información de enlace no es una referencia al objeto, sino la dirección de éste. Dicha dirección no se puede usar directamente, será necesario reconstruir el objeto a partir de ella.

Cuando el control se destruya deberemos restituir el anterior procedimiento de ventana, para lo cual bastará con llamar a *SetWindowLong* pasando el miembro **ProcedimientoAnterior** como último parámetro.

```
        ' Al destruir el control
Private Sub UserControl_Terminate()
        ' restablecemos su procedimiento original
    SetWindowLong hwnd, GWL_WNDPROC, Enlace.ProcedimientoAnterior
End Sub
```

Recuperación de los datos en el procedimiento de ventana

Una vez subclasificado el control, los mensajes comenzarán a llegar a la función **ProcesoMensaje** del módulo estándar. Es en este punto donde necesitamos recuperar la información de enlace. Para ello primero obtenemos la dirección de la estructura **PAXEnlace** usando la función *GetWindowLong*. La dirección obtenida la usamos con el procedimiento *RtlMoveMemory* para copiar los datos a una variable privada que hemos declarado, como puede verse en el siguiente fragmento.

```
 Private Declare Sub RtlMoveMemory Lib "kernel32" _
    (DestAddr As Any, SrcAddr As Any, ByVal Bytes As Long)

        ...
    Dim Direccion As Long ' recuperamos la dirección
```

```
Direccion = GetWindowLong(hwnd, GWL_USERDATA) ' de los datos de enlace

Dim Enlace As PAXEnlace ' variable de enlace vacía
    ' recuperamos la información original
RtlMoveMemory Enlace, ByVal Direccion, 8
```

Observe que se ha modificado la definición de la función *RtlMoveMemory*, de tal forma que los dos primeros parámetros se reciben por referencia. Esto evita tener que estar usando continuamente la función *VarPtr* para obtener la dirección y pasarla por valor. Al copiar los datos de enlace, sin embargo, es preciso pasar por valor la dirección que se ha recuperado con *GetWindowLong*, para lo cual simplemente se precede la variable **Direccion** de la palabra clave *ByVal*.

En este momento la variable privada **Enlace** ya tiene la dirección del control y la dirección del anterior procedimiento de ventana. En realidad lo que hemos hecho es copiar la variable privada **Enlace** declarada en el control, a nuestra propia variable privada. En caso de que el mensaje que se recibe esté relacionado con la palanca de juegos es necesario llamar al método **Evento**. Antes, sin embargo, deberemos reconstruir el objeto **PAXJoystick2** a partir de la dirección que hay en la variable **ControlAsociado**. Con este fin declaramos una variable **PAXJoystick2** capaz de almacenar una referencia, tras lo cual usamos *RtlMoveMemory* para copiar la dirección en su interior. A partir de este momento la variable apunta al control, por lo que podemos llamar sin ningún problema a sus métodos.

```
    ' Si es uno de los mensajes que nos interesa
If (Mensaje >= MM_JOY1BUTTONDOWN And Mensaje <= MM_JOY2BUTTONUP) Or _
   (Mensaje >= MM_JOY1MOVE And Mensaje <= MM_JOY2ZMOVE) Then
    Dim Objeto As PAXJoystick2 ' Variable sin ninguna referencia
        ' recuperamos la referencia al objeto original
    RtlMoveMemory Objeto, Enlace.ControlAsociado, 4
    Objeto.Evento Mensaje ' lo notificamos a nuestro cliente
    RtlMoveMemory Objeto, 0&, 4 ' eliminamos el puntero
End If
```

Objetos sin referencia

La variable **Objeto** declarada como local en la función **ProcesoMensaje** no tiene inicialmente referencia alguna, por lo que al llegar al final del código, momento en que se liberan las variables locales, Visual Basic sabe que sólo tiene que eliminar la variable. Si creamos un objeto y asociamos su referencia a esta variable, al salir de la función es necesario primero destruir el objeto, tras lo cual se elimina la variable. En un capítulo previo analizamos el mecanismo de control de referencias, consistente en un contador interno que se incrementa con cada asignación y se decrementa de forma automática cuando se eliminan referencias, de tal forma que si el contador llega al valor cero se destruye el objeto.

En el código anterior asignamos a la variable **Objeto** una referencia obtenida mediante un puntero. No se ha usado la instrucción *Set*, por lo cual el contador de referencias a ese objeto no se ha incrementado. Al llegar al final de la función, sin embargo, puesto que **Objeto** tiene una referencia, Visual Basic decrementará el contador y destruirá el objeto, es decir, nuestro control, que en este momento se está ejecutando. Un segundo paso por la función de proceso de mensajes será aún peor, puesto que al salir de ella se decrementará de nuevo el contador y se intentará destruir un objeto que ya no existe.

Para evitar estos problemas lo que tenemos que hacer es eliminar la referencia de la variable **Objeto** antes de llegar al final de la función. Por eso hacemos una nueva llamada a *RtlMoveMemory*, copiando en esa variable el valor cero.

Terminada la segunda versión del control puede insertarlo en un formulario y comprobar su correcto funcionamiento. Realmente en este caso sólo notaría la diferencia entre **PAXJoystick** y **PAXJoystick2** insertando dos controles en un formulario y usando dos palancas de juegos. El control **PAXJoystick** no sería capaz de funcionar correctamente, mientras que **PAXJoystick2** sí.

Miembros *Friend*

Otra de las novedades de la versión 5.0 de Visual Basic es la posibilidad de definir métodos *Friend* en el interior de módulos de clase y controles. La palabra *Friend* es un calificador como *Private* y *Public*, que afecta al ámbito en que es reconocido el método. Al declarar un método como *Friend* conseguiremos que sea público para todo el proyecto, pero privado para el exterior, es decir, una combinación entre los dos calificadores existentes hasta ahora.

En ocasiones es necesario usar desde un módulo de un proyecto métodos que se encuentran en otro. Si hacemos esos métodos públicos los podrá usar cualquiera, pero si los hacemos privados no se podrán usar nada mas que desde el

propio módulo. El control **PAXJoystick** cuenta con un método, llamado **Evento**, que se ha hecho público por la necesidad de ser llamado desde el módulo estándar, pero al tiempo se ha modificado uno de sus atributos para que permanezca oculto, puesto que no es un método candidato a ser usado por el usuario final del control.

Cuando nos encontramos con un caso así es mucho mejor definir el método como *Friend*. Cambiando *Public* por *Friend* en la declaración del método **Evento**, por ejemplo, conseguiremos que pueda ser usado por el módulo estándar pero, al tiempo, evitaremos que sea conocido fuera del proyecto, por lo que el usuario del control no podrá utilizarlo.

En otros lenguajes, como C++, también existe el calificador *friend*, aunque su comportamiento es diferente. Cuando se declara una clase, por ejemplo, se puede indicar qué otra clase o un cierto método tiene acceso a los miembros de la clase que se está declarando. Es decir, no se permite a todo el proyecto tener acceso, lo cual nos permite controlar perfectamente qué código podrá manipular el objeto. Desgraciadamente, en Visual Basic cuando se declara un método como *Friend* no es posible establecer limitaciones, el método es visible para todo el proyecto, sin excepción. Esto, sin embargo, no será un problema si todos los módulos del proyecto están estrechamente relacionados, que es lo habitual.

Resumiendo

Llegados al fin de este capítulo ya estamos preparados para comenzar a desarrollar controles de todo tipo. No sólo sabemos cómo definir propiedades de diferentes tipos, métodos y eventos, no solamente podemos crear controles que sepan como comportarse según el ambiente en el que están, además hemos aprendido a crear nuestras propias colecciones, a aprovechar cualquier función de la API de Windows que necesite punteros, a subclasificar nuestros controles, etc. A partir del próximo capítulo iremos poniendo en práctica todo lo aprendido con el fin de crear controles útiles, lo cual nos servirá, además, para conocer también nuevas funciones de la API de Windows y nuevos conceptos.