Microsoft

Visual Basic

Aplicaciones para Windows

Fco. Javier Ceballos Sierra
Profesor titular de la
Escuela Universitaria Politécnica
Universidad de Alcalá de Henares

ADDISON-WESLEY IBEROAMERICANA

ra-ma

ÍNDICE

PRÓLOGO XV

PARTE 1. VISUAL BASIC 1

CAPÍTULO 1. ¿QUÉ ES VISUAL BASIC? 3

INTRODUCCIÓN 3
COMO CREAR UNA APLICACIÓN 5
OTRAS FACILIDADES DE VISUAL BASIC 6
UN LENGUAJE DE ALTO NIVEL 9
MENÚS 10
COLORES 11
UTILIZANDO LA AYUDA 12
INSTALACIÓN DE VISUAL BASIC 13
Instalación desde Windows 13
Instalación desde MS-DOS 14
ARRANCANDO Visual Basic 14
Arranque desde Windows 14
Arranque desde MS-DOS 14
Opciones de la orden vb 14

CAPÍTULO 2. MI PRIMERA APLICACIÓN 15

INTRODUCCIÓN 15
PROGRAMANDO BAJO WINDOWS 17
DESARROLLO DE UNA APLICACIÓN 18
Crear una nueva aplicación 19

Mover y ajustar el tamaño de la forma 20
Dibujar los controles 21
Borrar un control 24
Propiedades de los objetos 24
Unir código a los objetos 27
Guardar la aplicación 30
Verificar la aplicación 31
Crear un fichero ejecutable 32
Cambio de propiedades en ejecución 32

CAPÍTULO 3. ELEMENTOS DEL LENGUAJE 35

INTRODUCCIÓN 35
COMENTARIOS 35
CONSTANTES NUMÉRICAS Y DE CARACTERES 36
VARIABLES 36
Nombres de variables 36
Tipos de datos 37
Ámbito de las variables 38
Variables locales 39
Variables a nivel de la forma y a nivel del módulo 39
Variables globales 40
Variables con el mismo nombre en diferentes niveles 40
OPERADORES 40
SENTENCIAS 41
PROPIEDADES 41
SENTENCIAS DE CONTROL 42
If ... Then ... Else 42
Select Case 43
For ... Next 45
Do ... Loop 46
PROCEDIMIENTOS Y FUNCIONES 47
Ámbito de un procedimiento 48
Crear un procedimiento general 48
Funciones (Function) 48
Procedimientos (Sub) 50
Declarar todas las variables locales como estáticas 51
Argumentos por referencia y por valor 51
Procedimientos recursivos 52
TIPOS DE DATOS Form y Control 52
Tipo de un control pasado como argumento 53

ARRAYS 54
Arrays de variables 55
Arrays dinámicos 56
Arrays de controles 56
EL MÓDULO GLOBAL 57
ESTRUCTURAS 58

CAPÍTULO 4. CONTROLES MÁS COMUNES 61

INTRODUCCIÓN 61
CONVERSIÓN DE TEMPERATURAS 61
DESARROLLO DE LA APLICACIÓN 62
Objetos 62
Sucesos 62
Pasos a seguir durante el desarrollo 62
La forma, los controles y sus propiedades 63
Escribir el código 64
Interceptando la tecla pulsada 66
CÓMO TRABAJA UN PROCEDIMIENTO 68
Recuperando el texto de las cajas de texto 68
Convirtiendo el texto a un número 68
Realizando cálculos 68
Dando formato a la salida 69
VISUALIZAR LA FECHA Y LA HORA 70
PASAR DE UN CONTROL A OTRO 72
BOTONES DE ÓRDENES 72
REEMPLAZANDO EL SEPARADOR DECIMAL 74
EJECUTAR ÓRDENES DEL SISTEMA 75

CAPÍTULO 5. ARRAYS DE CONTROLES 77

INTRODUCCIÓN 77
Creando controles en tiempo de ejecución 81
DISEÑO DE UNA CALCULADORA 83
Objetos 83
Sucesos 84
Pasos a seguir durante el desarrollo 84
Diseño de la forma y de los controles 84
Escribir el código 86

CAPÍTULO 6. TÉCNICAS DE DISEÑO 95
DISEÑO DE UN MENÚ 95
Propiedades de un menú 98
CAJA DE TEXTO CON MÚLTIPLES LÍNEAS 99
TRABAJAR CON TEXTO SELECCIONADO 100
UTILIZANDO EL PORTAPAPELES 101
DESARROLLO DE UN EDITOR 102
APLICACIONES CON MÚLTIPLES FORMAS 111
Métodos y sentencias para manipular formas 111
ASOCIAR UN ICONO A LA APLICACIÓN 112
EL FICHERO CONSTANT.TXT 112
MODIFICAR UN MENÚ EN TIEMPO DE EJECUCIÓN 113
DESARROLLO DE UN RELOJ DESPERTADOR 113
Temporizador 114
Diseño de la forma y de los controles 114
Unir el código a los controles y a la forma 116
Cambiar el [illegible] una orden de un menú 118
Añadir órdenes a un menú 120
Añadir una nueva forma 124
Procedimiento común para las órdenes añadidas 127
Borrar órdenes de un menú 128

CAPÍTULO 7. CAJAS DE DIÁLOGO 131

INTRODUCCIÓN 131
E/S DE DATOS CON InputBoxS y MsgBox 131
SEÑALES 134
OPCIONES 135
LISTAS Y COMBINADOS 138
Utilización de listas 139
Crear un módulo 144
Eliminar un elemento de una lista 145
Utilización de combinados 147
BARRAS DE DESPLAZAMIENTO 150
MARCOS 152
UN SHELL DE WINDOWS 152
COLORES 153
Función RGB 154
APLICACIONES A MEDIDA 155

CAPÍTULO 8. FICHEROS DE DATOS 161

INTRODUCCIÓN 161
TIPOS DE FICHEROS EN VISUAL BASIC 161
UTILIZACIÓN DE FICHEROS SECUENCIALES 162
Escribir en un fichero 164
Control de errores 165
CONTROLES RELATIVOS A FICHEROS 167
Utilización conjunta de estos controles 168
ABRIR UN FICHERO PARA LEER 169
Leer de un fichero secuencial 172
UTILIZACIÓN DE FICHEROS ALEATORIOS 176
Escribir en un fichero aleatorio 176
Leer de un fichero aleatorio 180
Trabajando directamente sobre el fichero 182
SENTENCIA Seek 187
FUNCIONES Loc y LOF 188
UTILIZACIÓN DE LA IMPRESORA 188
FICHEROS INDEXADOS 188

CAPÍTULO 9. EFECTOS GRÁFICOS 189

INTRODUCCIÓN 189
IMÁGENES 189
Cargar una imagen 189
OPERACIONES DINÁMICAS CON CONTROLES 193
DIBUJANDO CON VISUAL BASIC 196
Escribir texto 199
Gráficos persistentes 203
Limpiar el área de dibujo 204
Dibujar puntos 204
Dibujar líneas 207
Dibujar cajas 209
Dibujar círculos, elipses y arcos 211
Colorear figuras 213
Animación de gráficos 214

CAPÍTULO 10. SUCESOS DEL RATÓN 219
INTRODUCCIÓN 219
ARGUMENTOS DE LOS SUCESOS DEL RATÓN 219
Argumento Button 220
Argumento Shift 222
Argumentos X, Y 223
APLICACIONES GRÁFICAS 224
EJEMPLO DE UN PANEL DE DIBUJO 226
SELECCIONAR Y ARRASTRAR 238
EJEMPLO DE SELECCIONAR Y ARRASTRAR 243

CAPÍTULO 11. DEPURAR UNA APLICACIÓN 253
INTRODUCCIÓN 253
MANIPULACIÓN DE ERRORES 254
DEPURACIÓN 259
Ventana inmediata 261
CONSIDERACIONES ESPECIALES 262

PARTE 2. TÉCNICAS AVANZADAS 263

CAPÍTULO 12. FICHEROS INDEXADOS 265

INTRODUCCIÓN 265
ESTRUCTURA DE UN FICHERO SECUENCIAL INDEXADO 266
ALGORITMOS HASH 267
Tablas Hash 267
Método Hash abierto 268
Método Hash con overflow 270
Eliminación de elementos 271
PROCESO DE UN FICHERO SECUENCIAL INDEXADO 271
Acceso al fichero índice 276
IMPRESIÓN DE RESULTADOS 285

CAPÍTULO 13. INTERCAMBIO DINÁMICO DE DATOS 289

INTRODUCCIÓN 289
COMO TRABAJA EL DDE 289
CREAR UN ENLACE EN TIEMPO DE DISEÑO 291
Aplicación Visual Basic como cliente 291

Aplicación Visual Basic como servidor 294
CREAR UN ENLACE EN TIEMPO DE EJECUCIÓN 296
Aplicación Visual Basic como cliente 296
Establecer un enlace frío 301
Aplicación Visual Basic como servidor 302
SUCESOS ASOCIADOS CON LOS ENLACES 307
ENVIAR ORDENES A OTRAS APLICACIONES 310
ACTUALIZAR GRÁFICOS EN UN DDE 310
MANIPULACIÓN DE ERRORES 311
FUNCIÓN DoEvents 314
PEGAR VÍNCULOS Y COPIAR 317
Añadir la orden Pegar Vínculos 318
Añadir la orden Copiar 321
ENVIAR TECLAS A OTRAS APLICACIONES 322

CAPÍTULO 14. LLAMADAS A LAS FUNCIONES DE LA API DE WINDOWS 325

INTRODUCCIÓN 325
LIBRERÍAS DINÁMICAS 325
DECLARACIÓN DE UNA FUNCIÓN DE UNA DLL 326
LLAMADA A UNA FUNCIÓN DE UNA DLL 328
TIPOS DE DATOS EN LAS LLAMADAS 328
Cadenas de caracteres (Strings) 328
Arrays 329
Tipos definidos por el usuario (estructuras) 330
Punteros nulos 330
Handles 331
Propiedades 332
Formas y controles 333
TIPOS DE WINDOWS 333
Cómo pasar un valor a un parámetro de tipo BYTE 333
MENSAJES 334
EDITOR DE TEXTOS 335
BORRAR UNA LISTA 344
PALABRA DE PASO 346
MOVIENDO CONTROLES ENTRE FORMAS 349
IMPRIMIR UNA IMAGEN VISUAL BASIC 354

PARTE 1

Visual Basic

- ¿Qué es Visual Basic?
- Mi primera aplicación
- Elementos del lenguaje
- Controles más comunes
- Arrays de controles
- Técnicas de diseño
- Cajas de diálogo
- Ficheros de datos
- Efectos gráficos
- Sucesos del ratón
- Depurar una aplicación

CAPÍTULO 1

¿QUÉ ES VISUAL BASIC?

INTRODUCCIÓN

Visual Basic es un sistema de desarrollo diseñado especialmente para crear aplicaciones gráficas de una forma rápida y sencilla. Incluye una utilidad para diseños gráficos y un lenguaje de alto nivel.

Al escribir este libro el autor ha supuesto que el lector conoce los conceptos fundamentales para utilizar Windows y que posee conocimientos de programación. Si no es así, empiece por adquirir conocimientos sobre las sentencias y estructuras básicas de un lenguaje de programación; para ello le recomiendo mi libro *Curso de programación QBASIC y MS-DOS 5* publicado por la editorial RA-MA. En cambio, si ya tiene conocimientos de QBASIC o de QuickBASIC, le resultará mucho más fácil aprender a desarrollar programas con Visual Basic.

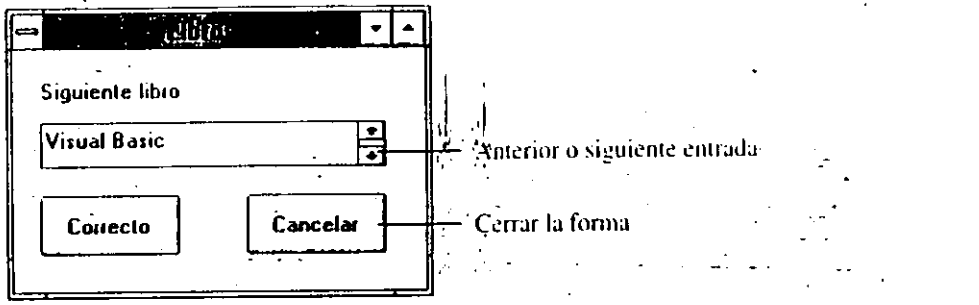
Visual Basic está centrado en dos tipos de objetos, *ventanas* y *controles*, que permiten diseñar sin programar, un mecanismo de comunicación (interfaz) para una aplicación. Para realizar una aplicación se crean *ventanas*, llamadas *formas*, y sobre ellas se dibujan otros objetos llamados *controles*. A continuación se escribe el código fuente relacionado con cada objeto.

Quiere esto decir, que cada *objeto* (*formas* y *controles*) está ligado a un código que permanece inactivo hasta que se dé el suceso que lo activa. Por ejemplo, podemos programar un botón (objeto que se puede pulsar) que responda a un clic del ratón.

Visual Basic también incluye,

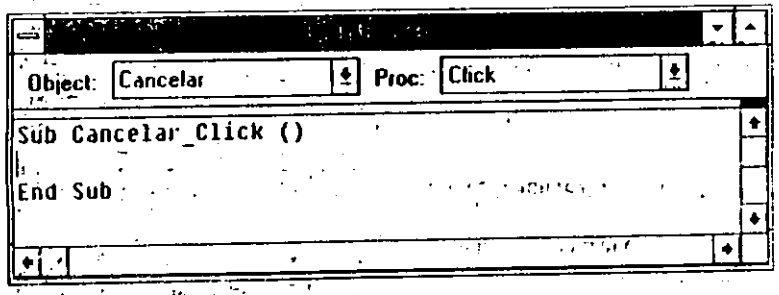
- soporte para intercambio de datos con otras aplicaciones (DDE - **dynamic data exchange**)

Para hacer que una aplicación responda a sucesos tenemos que unir código a las formas y a los controles.



El código que se une a un objeto se denomina *procedimiento conducido por un suceso* o simplemente *procedimiento*.

Para unir un procedimiento a un objeto hay que escribir el código correspondiente a dicho procedimiento en la ventana de código asociada con el objeto. Para visualizar la ventana de código de una determinada forma, hay que hacer un doble clic dentro del área correspondiente a esa forma y haciendo un doble clic sobre el área correspondiente a un control, se visualiza la ventana de código para ese control.



Un procedimiento está formado por un nombre de objeto, un nombre de suceso y una o más sentencias. La figura anterior muestra el esquema para el procedimiento asociado con el control *Cancelar*. Observe que el nombre del objeto es *Cancelar*, el nombre del suceso es *Click* y el nombre del procedimiento es *Cancelar_Click*.

Todo el código asociado con una forma y con sus controles se almacena con esa forma. Cuando una aplicación consta de varias formas y se quiere compartir código entre ellas, hay que poner el código en un módulo.

Un *módulo* es un fichero que contiene los procedimientos que son compartidos por las formas de una aplicación.

UN LENGUAJE DE ALTO NIVEL

Visual Basic incluye un lenguaje fácil de utilizar que proporciona muchas de las características de Microsoft QBasic o de Microsoft QuickBasic y de Microsoft Basic Compiler.

Algunas de ellas son:

- Bloques IF ... THEN ... ELSE

```
If var1 > 0 And var2 > 0 Then
   var3 = var2 / var1
Else
   var3 = 0
End if
```

- Bucles

```
Do While i <= 100
   Print i
   i = i + 1
Loop
```

- Ocho tipos de datos

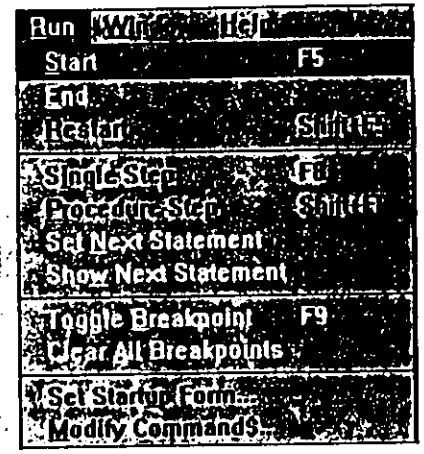
```
Integer, Long, Single, Double
Currency, String, Form, Control
```

- Funciones matemáticas y de cadena

```
Abs(var1 - var2)
Print Ucase$(libro$)
```

Como ayuda para revisar el código rápidamente, Visual Basic proporciona:

- Verificación automática de la sintaxis.
- Utilidades para depurar código



Órdenes para ejecución

Ejecución paso a paso

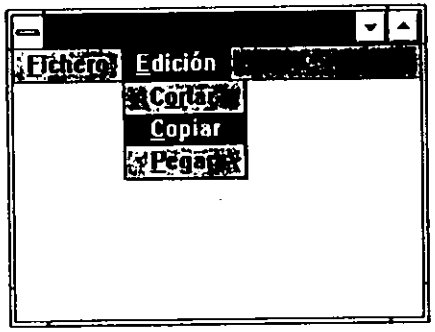
Puntos de parada

Condiciones de arranque

- Ventana inmediata que permite probar una línea de código o verificar una variable.

MENÚS

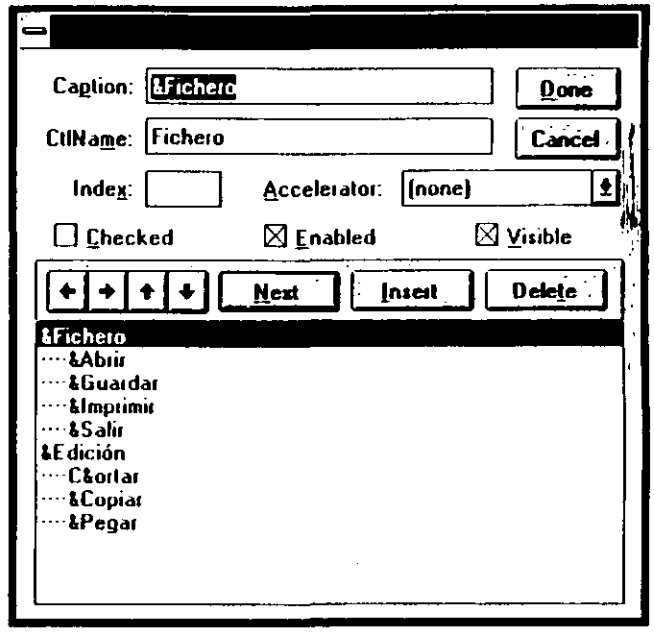
Cuando en una aplicación se utilizan muchas órdenes, Visual Basic permite agruparlas en un menú.



Un menú se diseña utilizando la ventana de diseños de menús.

Para visualizar esta ventana hay que ejecutar la orden **Menu Design Window** del menú **Window**.

Como se ve en la figura siguiente, el nombre del menú que aparecerá en la barra de menús, se escribe en la caja de texto **Caption**.



Para resaltar la letra que da acceso al menú se inserta un ampersand (&) antes de la misma. El usuario podrá seleccionar este menú pulsando la tecla **Alt** y la tecla correspondiente a la letra resaltada, la cual aparece subrayada. También puede realizar la selección utilizando el ratón.

COLORES

Visual Basic permite también añadir colores a los objetos. La paleta de colores ofrece 48 colores estándar. También se pueden definir colores adicionales pulsando el botón **Custom Colors >>** que aparece en la ventana que se presenta a continuación.

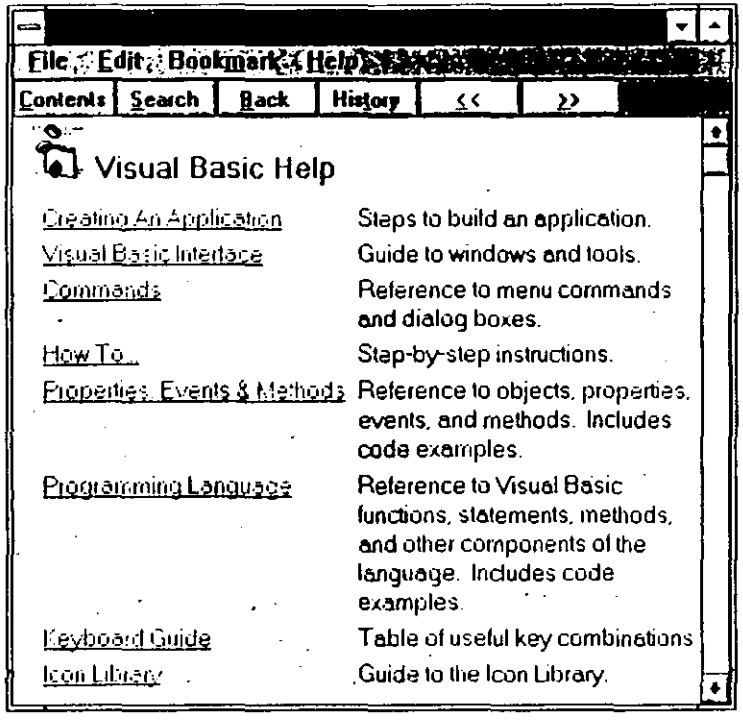
1. Muestra el color actual del primer plano (cuadrado interno) y el color de fondo (cuadrado externo).
2. Muestra el color del texto.

3. Pone al objeto seleccionado, los colores utilizados por defecto por Visual Basic.

UTILIZANDO LA AYUDA

Visual Basic tiene instalada una ayuda en línea que da información acerca de propiedades, sucesos, mensajes de error, sentencias, teclado, depurador, intercambio dinámico de datos y entorno de Visual Basic.

Para obtener ayuda, puede utilizar el índice proporcionado por el menú **Help** de la barra de menús de Visual Basic, o puede seleccionar el elemento deseado en el entorno de Visual Basic y pulsar **F1**.



Cuando esté dentro de la ayuda,

- utilice la orden **Search** para buscar un término o una palabra clave del lengu

- haga clic sobre una frase o palabra subrayada para ver la ayuda referente a la misma, o
- haga clic sobre una palabra subrayada con trazos discontinuos para ver su definición.

Cuando se visualice un mensaje y desee ayuda sobre el mismo, pulse **F1**.

Cuando esté dentro de la ventana de código, seleccione una palabra del lenguaje y pulse **F1** para obtener ayuda sobre la misma.

Para obtener ayuda sobre un menú o una herramienta de Visual Basic, selecciónela y pulse **F1**.

INSTALACIÓN DE VISUAL BASIC

Antes de instalar Visual Basic compruebe que su ordenador cumple los requisitos mínimos. Visual Basic para Windows 3.0 o posterior, corre en modo estándar (286) o en modo extendido (386). Los requerimientos básicos son:

- Un ordenador con microprocesador 80286 o superior.
- Un disco duro.
- Un ratón.
- Una placa de video CGA, EGA, VGA, 8514, Hércules, o compatibles (se recomienda EGA o superior).
- MS-DOS versión 3.1 o superior.
- Windows versión 3.0 o superior.
- 1 Mb de memoria o más.

También se recomienda leer el fichero README.TXT.

Si su sistema cumple los requerimientos mínimos, puede comenzar a instalar Visual Basic. La instalación puede realizarse desde Windows o desde MS-DOS.

Instalación desde Windows

1. Inserte el disco 1 en la unidad A.
2. Active el administrador de archivos y ejecute la orden **Ejecut** un) del menú **Archivo** (File).

3. Escriba en la caja de texto **a:setup**
4. A continuación siga las instrucciones dadas en la pantalla.

Instalación desde MS-DOS

1. Inserte el disco 1 en la unidad A.
2. Escriba **a:**
3. Escriba **win setup**
4. A continuación siga las instrucciones dadas en la pantalla.

ARRANCANDO Visual Basic

Visual Basic se puede arrancar desde Windows o desde MS-DOS.

Arranque desde Windows

1. Active el administrador de archivos y ejecute la orden **Ejecutar** (Run) del menú **Archivo** (File).
2. Escriba en la caja de texto **vb**

Arranque desde MS-DOS

- Escriba a continuación del símbolo de sistema **win vb**

Opciones de la orden vb

vb [[/run] *fichero*][/cmd *argumentos*

/run permite que se ejecute la aplicación Visual Basic especificada por *fichero*.

/cmd permite pasar argumentos en la línea de órdenes que serán puestos a disposición de la aplicación Visual Basic, por la función **Command$**.

```
If Command$ = "" Then
  Msg$ = "No se han pasado argumentos"
```

CAPÍTULO 2

MI PRIMERA APLICACIÓN

INTRODUCCIÓN

Bienvenido a Visual Basic, un potente paquete perteneciente a la nueva generación de compiladores para el desarrollo de aplicaciones. La programación bajo Windows con Visual Basic permite desarrollar aplicaciones para Windows o para OS/2 Presentation Manager.

Según se ha expuesto en el capítulo anterior, Visual Basic es una caja inmensa de utilidades y recursos puestos a nuestra disposición para hacer que la programación de cualquier aplicación por compleja que sea, resulte ahora más sencilla que nunca y con pantallas infinitamente mejor presentadas e intuitivas.

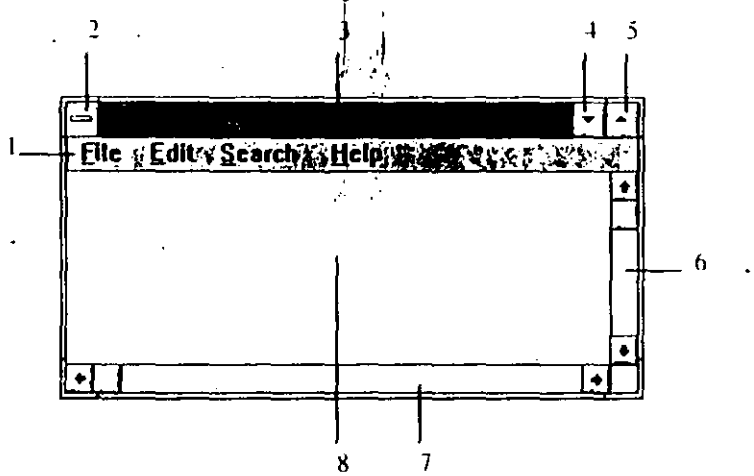
Sin lugar a dudas el desarrollo de aplicaciones bajo Windows es una realidad, lo cual hace suponer que el futuro de la microprogramación va encaminado hacia los entornos gráficos.

Una de las características fundamentales que diferencian a Windows de DOS es un entorno gráfico que a base de ventanas presenta al usuario en forma de objetos, la mayoría de las operaciones que se pueden realizar bajo el sistema operativo, evitando así tener que recordar órdenes y opciones que en muchos casos el escribirlas ya eran una posible fuente de errores. En un entorno gráfico un usuario simplemente selecciona la utilidad correspondiente a la tarea que desea ejecutar; el resto del trabajo lo hace el sistema.

Una de las grandes ventajas de trabajar con Windows es que todas las ventanas se comportan de la misma forma y todas las aplicaciones utilizan los mismos métodos básicos (menús descendentes, botones) para introducir órdenes.

Una ventana típica de Windows tiene las siguientes partes:

1. *Barra de menús*. Visualiza el conjunto de los menús disponibles para esa aplicación. Cuando alguno de los menús se activa haciendo clic con el ratón sobre su título, se visualizan el conjunto de órdenes que lo forman.



2. *Menú de control*. El menú de control proporciona órdenes para: restaurar a su tamaño, mover, dimensionar, minimizar, maximizar y cerrar, la ventana. Incluye también la orden **Cambiar a...** que invoca la lista de tareas activadas.
3. *Barra de título*. Contiene el nombre de la ventana.
4. *Botón para minimizar la ventana*. Cuando se pulsa este botón la ventana se reduce a un icono. Esta es la mejor forma de mantener las aplicaciones, cuando tenemos varias de ellas activadas y no se están utilizando en ese instante.
5. *Botón para maximizar la ventana*. Cuando se pulsa este botón la ventana se amplia al máximo y el botón se transforma en otro con dos flechas. Si éste se pulsa, la ventana se reduce al tamaño anterior.
6. *Barra de desplazamiento vertical*. Cuando la información no entra verticalmente en una ventana, Windows añade una barra de desplazamiento vertical a la derecha de la ventana.
7. *Barra de desplazamiento horizontal*. Cuando la información no entra horizontalmente en una ventana, Windows añade una barra de desplazamiento horizontal en el fondo de la ventana.

La barra de desplazamiento tiene un cuadrado que se desplaza a lo largo de [illegible] para indicar en qué posición nos encontramos con respecto al principio y al final de la información tratada.

Para desplazarse:

- Una línea verticalmente o un carácter horizontalmente, utilice las flechas de los extremos de las barras.
- Varias líneas verticalmente o varios caracteres horizontalmente, pulse con el ratón una flecha y mantenga el botón pulsado.
- Aproximadamente una pantalla completa, haga clic sobre la barra de desplazamiento. Para subir haga clic encima del cuadrado de la barra vertical y para bajar haga clic debajo del cuadrado. Para moverse a la izquierda, haga clic a la izquierda del cuadrado de la barra horizontal y para moverse a la derecha haga clic a la derecha del cuadrado.
- A un lugar específico, haga clic sobre el cuadrado y manteniendo el botón pulsado arrastre el cuadrado.

8. *Área del usuario*. Es la parte de la ventana en la que el usuario coloca el texto y los gráficos.

Un objeto en general, puede ser movido a otro lugar haciendo clic sobre él y arrastrándolo manteniendo pulsado el botón izquierdo del ratón.

PROGRAMANDO BAJO WINDOWS

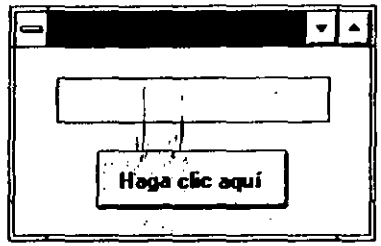
La programación que realizamos bajo DOS, por ejemplo con QBasic, difiere bastante de cómo se programa una aplicación en Windows, por ejemplo con Visual Basic. Un programa escrito bajo DOS es un conjunto de sentencias que se ejecutan de arriba a abajo, mas o menos, en el orden que el programador ha diseñado. Por ejemplo,

```
PRINT "Pulse una tecla para continuar ..."
CLS
WHILE INKEY$ = "": WEND
PRINT "Bienvenido a QBasic"
```

En este ejemplo la sentencia WHILE simplemente espera hasta que el usuario pulse una tecla, momento en el cual se pasa a la línea siguiente y aparece el mensaje *Bienvenido a QBasic*. Si hubiera más sentencias a continuación, la ejecución continuaría secuencialmente.

Una aplicación bajo Windows presenta todas las opciones posibles en una o más formas para que el usuario elija una de ellas. Esto da lugar a una nueva forma de pensar y de programar. Por ejemplo, en la figura siguiente, el usuario haga clic sobre el botón *Haga clic aquí*, en la caja de texto aparecerá un mensaje

Bienvenido a Visual Basic. Se dice entonces que la programación es *conducida por sucesos* y *orientada a objetos*.



Cuando desarrollamos una aplicación bajo este tipo de programación, la secuencia en la que van a ejecutarse las sentencias no puede ser prevista por el programador. Por ejemplo, si en lugar de un botón hubiera dos o más botones, claramente se ve que el programador no puede escribir el programa pensando que el usuario va a pulsarlos en una determinada secuencia.

Por lo tanto, para programar una aplicación Windows hay que escribir código separado para cada objeto en general, quedando la aplicación dividida en pequeños procedimientos conducidos cada uno de ellos por un suceso. Por ejemplo,

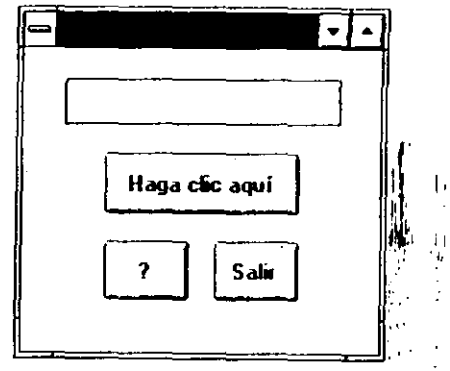
```
Sub Orden_Click ()
  Mensaje.Text = "Bienvenido a Visual Basic"
End Sub
```

El procedimiento *Orden_Click* es puesto en ejecución por el suceso *Click* y está ligado al objeto que hemos denominado *Orden* (botón titulado "Haga clic aquí"). Por ésto, esta forma de programar se denomina programación conducida por sucesos y orientada a objetos. Las ventanas, los botones, las cajas de texto, etc., son para nosotros objetos que, como veremos a continuación, tienen asociado un conjunto de datos y un código.

Esta nueva forma de desarrollar una aplicación se aparta bastante de la programación bajo DOS donde el programa era un solo bloque que se leía de arriba a abajo.

DESARROLLO DE UNA APLICACIÓN

Juntos y haciéndolo fácil vamos a construir nuestra primera aplicación. Esta tendrá una ventana (forma), una caja de texto y tres botones.



En esta aplicación, cuando el usuario pulse el botón "Haga clic aquí", en la caja de texto tiene que aparecer el mensaje "Bienvenido a Visual Basic", cuando pulse el botón "?" en la caja de texto tiene que aparecer el mensaje "¿Contento?" y cuando pulse el botón "Salir", la aplicación finalizará.

Para comenzar arrancamos Visual Basic como se indicó en el capítulo anterior.

En general, para construir una aplicación siga los pasos indicados a continuación:

1. Cree una nueva aplicación.
2. Mueva y ajuste el tamaño por defecto de la forma.
3. Dibuje los controles.
4. Defina las propiedades de la forma y de los controles.
5. Escriba el código para cada uno de los objetos.
6. Guarde la aplicación.
7. Verifique la aplicación.
8. Cree un fichero ejecutable.

Crear una nueva aplicación

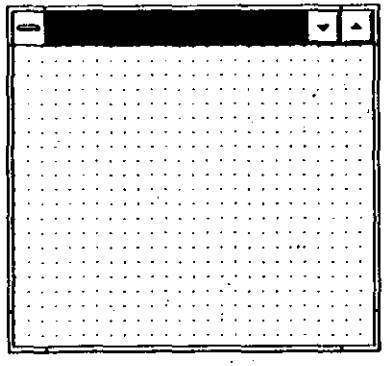
Este paso se ejecuta automáticamente cuando se arranca Visual Basic. En otro caso, esto es, se ha finalizado con una aplicación y se desea empezar otra nueva, ejecutaremos la orden **New Project** del menú **File**.

Una aplicación Visual Basic puede estar formada por tres clases de ficheros: formas, módulos, y un módulo global. Una forma es un fichero que contiene objetos gráficos, llamados controles, más código, mientras que un módulo contiene solamente código.

En este momento en el centro de la pantalla hay una ventana titulada *Form1*. Visual Basic denomina a las ventanas *formas*. Observe cómo la forma tiene algunos de los elementos comunes a las ventanas de Windows descritas anteriormente.

Mover y ajustar el tamaño de la forma

La forma es el plano de fondo para los controles. Se puede cambiar el tamaño y la situación de la misma utilizando el ratón. Para cambiar el tamaño apunte con el ratón a uno de los lados o a una de las esquinas de la forma, y cuando el puntero cambie a una flecha doble, con el botón izquierdo del ratón pulsado, arrastre en el sentido adecuado hasta conseguir el tamaño deseado.



En la figura se observa que el área del usuario está llena de puntos regularmente distribuidos. Estos puntos forman una rejilla que nos ayudará a alinear los controles que coloquemos dentro de la forma. Para hacer que la rejilla desaparezca ejecutaremos la orden **Grid Settings...** del menú **Edit** y desactivaremos la opción **Show Grid** (mostrar rejilla) haciendo clic sobre el cuadrado adjunto. Para finalizar se hace clic sobre el botón **OK**.

Si ahora le decimos a Visual Basic que ejecute este programa, para lo cual tendremos que ejecutar la orden **Start** del menú **Run**, la ventana sin la rejilla aparecerá sobre la pantalla y podremos actuar sobre cualquiera de sus controles (mover, maximizar, mover, ajustar el tamaño, etc.). Esta es la parte que Visual Basic realiza por nosotros y para nosotros: pruébelo. Para finalizar dispone de varias posibilidades:

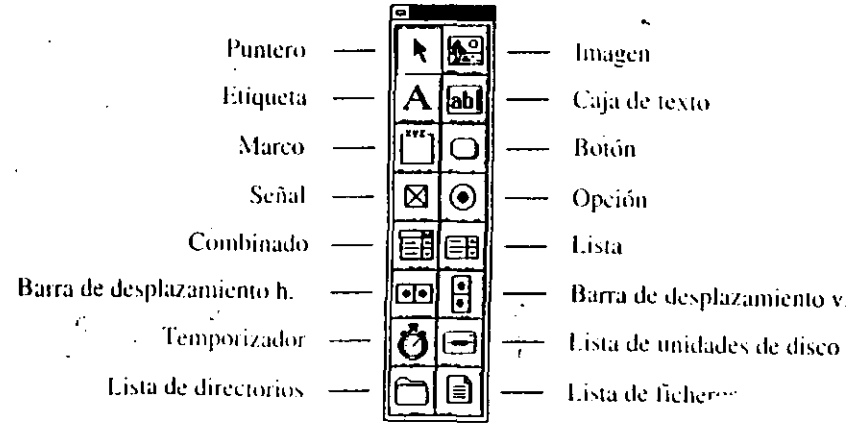
1. Haga un doble clic sobre el botón del menú de control de *Form1*.
2. Active el menú de control de la ventana *Form1* y ejecute **Close**.
3. Pulse las teclas **Alt + F4**.
4. Ejecute la orden **End** del menú **Run**.

Dibujar los controles

Hay dos tipos de objetos en Visual Basic: *formas* y *controles*. Los controles son objetos gráficos que nosotros dibujamos sobre una forma, tales como cajas de texto, botones, etiquetas, marcos, listas y temporizadores. Una forma es una ventana sobre la que nosotros diseñamos los elementos que el usuario tiene que utilizar para comunicarse con la aplicación. Los elementos son los controles, esto es, objetos que permiten entrar o salir datos. La forma más los controles forman la interfaz o medio de comunicación.

Si usted está familiarizado con la programación orientada a objetos, recordará que un objeto no solamente tiene asociado datos, sino también procedimientos. En C++, estos procedimientos reciben el nombre de funciones miembro. En otros lenguajes, como en Visual Basic, reciben el nombre de *métodos*.

Para añadir un control a una forma utilizaremos el panel de utilidades. Cada utilidad del panel crea un único control. La figura siguiente muestra este panel.



El *puntero* se utiliza para manipular los controles existentes sobre la forma. Con el puntero se puede seleccionar, mover y ajustar el tamaño de los objetos.

Una caja de *imagen* se utiliza cuando queremos visualizar una imagen que nosotros dibujamos utilizando código o que importamos de algún fichero.

Utilizaremos una *etiqueta* cuando queramos un texto, de una o más líneas, que no pueda ser modificado por el usuario. Una etiqueta se utiliza para dar instrucciones al usuario.

Una *caja de texto* es un área dentro de la forma en la que el usuario puede escribir o visualizar texto.

Un *marco* se utiliza para realzar el aspecto de la forma. A veces utilizamos los marcos para agrupar los objetos relacionados entre sí. Los marcos tienen propiedades propias como, por ejemplo, color.

Un *botón* tiene asociada una orden con él. Esta orden se ejecutará cuando el usuario haga clic sobre el botón.

Una *señal* se utiliza para seleccionar una opción. De esta forma se pueden seleccionar varias opciones de un grupo.

El control *opción* se utiliza para seleccionar una opción de entre varias. De esta forma sólo se puede seleccionar una opción de un grupo de ellas.

El control *combinado* combina una caja de texto y una lista.

El control *lista* contiene una lista enrollable de la que el usuario puede seleccionar un elemento.

La *barra de desplazamiento horizontal* permite desplazar la información de una caja hacia la izquierda o hacia la derecha.

La *barra de desplazamiento vertical* permite desplazar la información de una caja hacia arriba o hacia abajo.

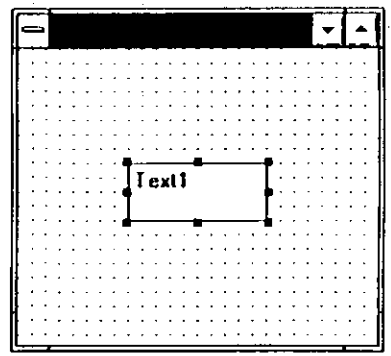
El *temporizador* permite activar procesos a intervalos regulares de tiempo.

La *lista de unidades de disco* se utiliza para visualizar la lista de unidades disponibles, con el fin de seleccionar una.

La *lista de directorios* se utiliza para visualizar los directorios a los que el usuario puede moverse.

La *lista de ficheros* se utiliza para visualizar los ficheros de un determinado directorio a los que el usuario puede acceder.

Siguiendo con nuestra aplicación, seleccionados del panel de utilidades que acabamos de describir, los controles que vamos a utilizar. En primer lugar vamos a crear una caja de texto. Para ello, hacemos un doble clic sobre la utilidad caja de texto, y vemos como en el centro de la forma aparece una caja de texto de un tamaño predefinido, según se muestra en la figura siguiente.

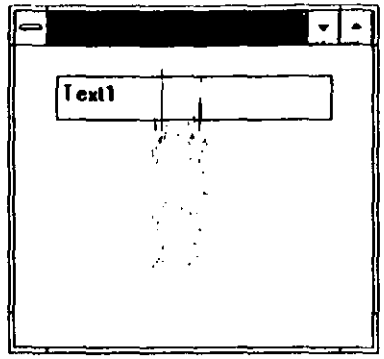


Observe que en la barra de propiedades, comentada en el capítulo anterior, se visualiza la propiedad **CtlName** (nombre del control) que tiene asignado el valor *Text1* que es el nombre dado al control por defecto. El nombre del control se utiliza para referirnos al control en nuestro programa.

También se observa sobre la caja de texto ocho cuadrados pequeños de color negro que reciben el nombre de *modificadores de tamaño*, los cuales permiten modificar el tamaño de los controles que estamos diseñando. Para modificar el tamaño de un control, primero selecciónelo haciendo clic sobre él, después apunte con el ratón a alguno de los modificadores de tamaño, observe que aparece una doble flecha, entonces, con el botón izquierdo del ratón pulsado arrastre en el sentido que desee ajustar el tamaño.

También puede mover el control a un lugar deseado dentro de la forma. Para mover un control, primero selecciónelo haciendo clic sobre él y después apunte con el ratón a alguna zona perteneciente al mismo y con el botón izquierdo del ratón pulsado arrastre hasta situarlo en el lugar deseado.

Ahora ajuste el tamaño de la caja de texto y muévala para que se corresponda con la figura que se presenta a continuación.



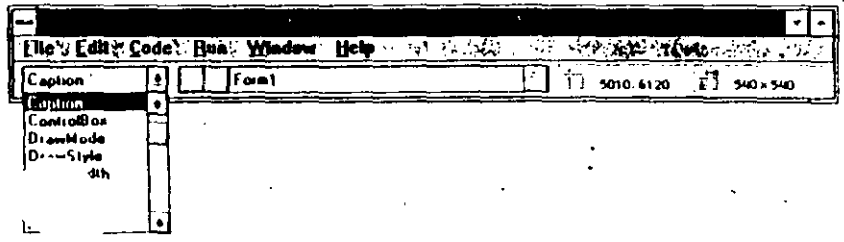
Borrar un control

Para borrar un control, primero se selecciona haciendo clic sobre él y a continuación se pulsa la tecla **Del** (Supr). Para borrar dos o más controles primero selecciónelos haciendo clic sobre cada uno de ellos al mismo tiempo que mantiene pulsada la tecla **Ctrl**, y después pulse **Del**.

Propiedades de los objetos

Cada tipo de objeto tiene ciertas propiedades como título, nombre, color, etc. Las propiedades de un objeto representan todos los datos que por definición están asociados con ese objeto.

Para cambiar las propiedades de un objeto tenemos que utilizar la barra de propiedades que se muestra en la figura siguiente y que se corresponde con la barra que está debajo de la barra de menús. En esta figura se observa en la lista de la izquierda las propiedades de la forma titulada *Form1*.



Cuando se selecciona un objeto puede verse la lista entera de propiedades, haciendo clic sobre la flecha de la barra de propiedades situada a la derecha de **Caption** (ver figura anterior).

Algunas propiedades son comunes a varios objetos y otras son únicas para un objeto determinado. Por ejemplo, la propiedad **Caption** (título) la tienen varios objetos, pero la propiedad **Interval** sólo la tiene el temporizador.

Cada propiedad de un objeto tiene un valor por defecto que puede ser modificado si se desea. Por ejemplo, la propiedad **Caption** de la forma del ejemplo que nos ocupa tiene el valor *Form1*.

Para verificar el valor de una determinada propiedad correspondiente a varios objetos, se selecciona ésta en la barra de propiedades para uno de ellos, y a continuación se pasa de un objeto al siguiente haciendo clic con el ratón sobre cada objeto, o simplemente pulsando la tecla **Tab**.

Para cambiar el valor de una propiedad de un objeto siga los pasos indicados a continuación:

1. Seleccione el objeto. Para ello, haga clic sobre el objeto o pulse sucesivamente la tecla **Tab** hasta que esté seleccionado. Un control seleccionado aparece rodeado por los modificadores de tamaño y una forma seleccionada cambia el color de fondo del título.

2. Seleccione de la lista de propiedades la propiedad que desea cambiar.

3. Modifique el valor que actualmente tiene la propiedad seleccionada. El valor actual de la propiedad seleccionada puede verse en la caja de la barra de propiedades, situada a continuación de la caja de propiedades. Para cambiar el valor escriba directamente sobre la caja que presenta el valor actual o, si es posible, seleccione uno de la lista que se despliega haciendo clic sobre la flecha que aparece a la derecha de esta caja. Para algunas propiedades, esta flecha es sustituida por tres puntos (...). En este caso se visualizará un cuadro de diálogo.

Otra forma de modificar una propiedad es durante la ejecución de la aplicación. Esto implica añadir el código necesario al procedimiento que deba realizar la modificación.

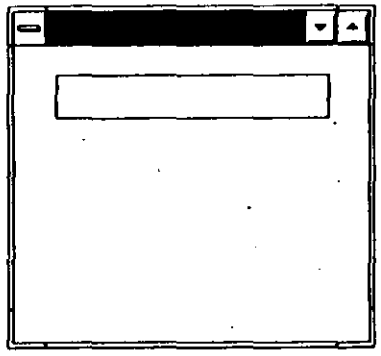
Siguiendo con nuestro ejemplo vamos a cambiar el título *F* de la forma, por el título *Mi primera aplicación*. Para ello, seleccione la f a continuación la propiedad **Caption** en la lista de propiedad . Despu úese en la se-

gunda caja de la barra de propiedades haciendo clic sobre ella, y sobrescriba el texto "Form1" con el texto "Mi primera aplicación"; finalice la operación haciendo clic sobre ☑ o pulsando **Enter**.

Veamos ahora las propiedades de la caja de texto. Seleccione la caja de texto y despliegue la lista de propiedades como se ha descrito anteriormente. Algunas de estas propiedades son **BackColor** (color del fondo de la caja de texto); **CtlName** (nombre dado a la caja de texto para referirnos a ella en el código); y **Text** (contenido inicial de la caja de texto).

Siguiendo los pasos descritos anteriormente, cambie el valor actual de la propiedad **CtlName** al valor *Mensaje* y el valor *Text1* de la propiedad **Text** a nada (esto es borre *Text1*). A continuación vamos a cambiar el color de fondo de la caja de texto. Para ello, seleccione la propiedad **BackColor** de la lista de propiedades, pulse el botón [...] situado a la derecha de la segunda caja de la barra de propiedades, elija como color de fondo (ver colores en el capítulo anterior) el quinto color de la tercera línea y cierre la ventana de colores (**Alt+F4**).

El resultado se muestra en la figura siguiente.



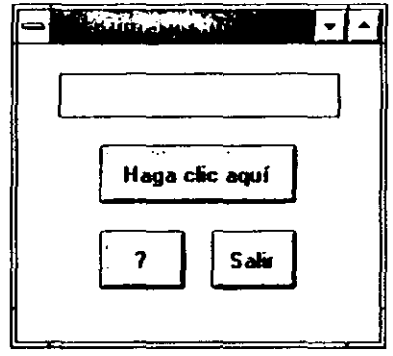
Nuestro paso siguiente es añadir los controles que nos faltan. Recordando el enunciado de la aplicación, tenemos que agregar tres botones de órdenes.

Para añadir el botón primero, hacemos un doble clic sobre la utilidad botón del panel de utilidades. Observamos que en el centro de la forma aparece un botón titulado **Command1**. Vuelva unas hojas atrás en este mismo capítulo y mire como tienen que quedar finalmente la forma y los controles. Mueva el botón y ajuste su tamaño para conseguir los mismos resultados a los que acabamos de hacer referencia. Ahora modifique las propiedades y asigne a **Caption** (título) el valor *Haga clic aquí*, y a **CtlName** (nombre) el valor *Orden1*. Siguiendo los

mismos pasos, añada los botones que quedan y asígneles las propiedades indicadas en el cuadro que se presenta a continuación.

	Caption	CtlName
Botón segundo	?	Orden2
Botón tercero	Salir	Orden3

El resultado que se obtiene después de todas estas operaciones es el siguiente:



Unir código a los objetos

Sabemos que el nombre de un objeto (**CtlName**) nos permite referirnos a él dentro del código de un programa. Por ejemplo, a la caja de texto se le ha dado el nombre *Mensaje*. Pues bien, para poder asignar dentro de un procedimiento el valor *Bienvenido a Visual Basic* a la propiedad **Text** del objeto *Mensaje* escribiremos,

```
Mensaje.Text = "Bienvenido a Visual Basic"
```

En otras palabras, la forma general en Visual Basic de referirse a una propiedad de un determinado objeto es:

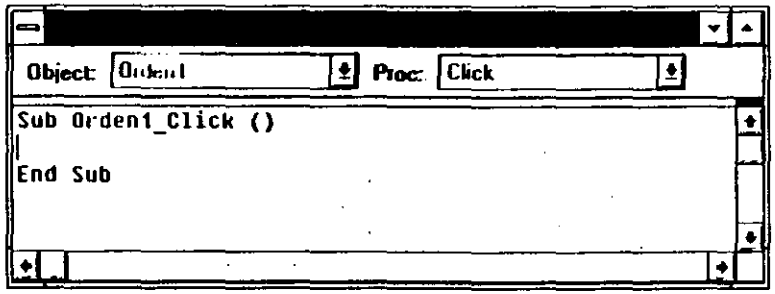
Objeto.Propiedad

donde *Objeto* es el nombre (**CtlName**) del objeto, forma o control, que tiene esa propiedad y *Propiedad* es el nombre de la propiedad del mismo.

Una vez que hemos creado la interfaz o medio de comunicación entre la aplicación y el usuario, tenemos que unir el código correspondiente a cada uno de los objetos para hacer que la aplicación responda a las acciones del usuario.

Como usted sabe, una aplicación bajo Windows es conducida por *sucesos* y *orientada a objetos*. Esto significa que nosotros ligamos el código escrito para un determinado objeto a un suceso que puede ocurrir sobre dicho objeto, de tal forma que cuando ocurra el suceso, dicho código se ejecute. Por ello, la unidad que agrupa ese código recibe el nombre de *procedimiento conducido por un suceso*.

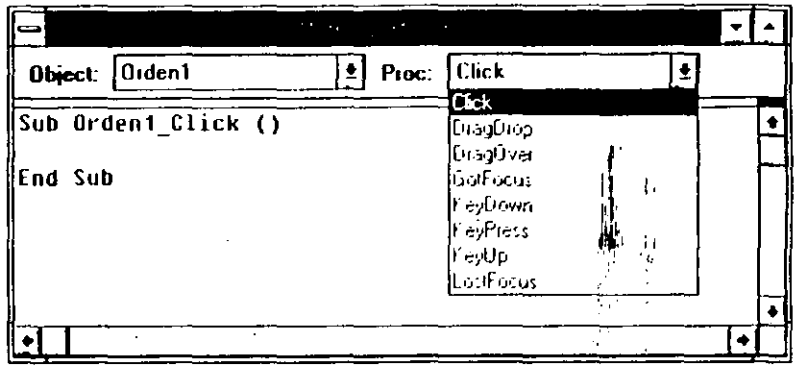
Mientras diseñamos un programa, para ver los posibles sucesos que se pueden asociar con un objeto se hace un doble clic sobre el objeto. Aparece entonces una nueva ventana, llamada *ventana de código*, como se muestra en la figura siguiente.



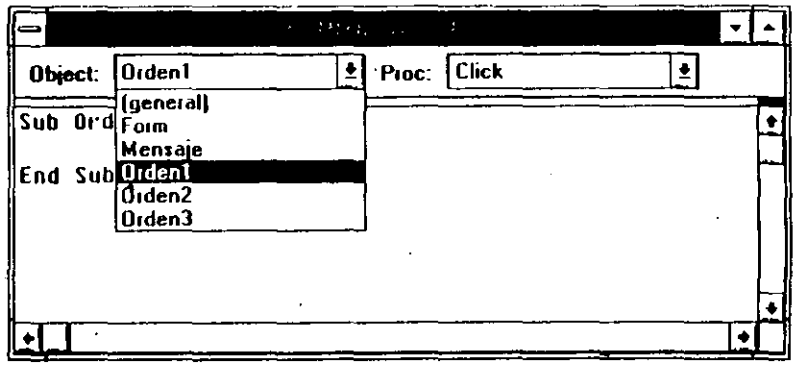
La ventana de código tiene un esquema para cada procedimiento conducido por un suceso. La figura anterior presenta el esquema para el procedimiento *Orden1_Click* conducido por el suceso *Click*. En otras palabras, el nombre del procedimiento *Orden1_Click* indica que ese procedimiento conduc[illegible] el suceso *Click* está conectado con el botón *Orden1*, y que será ejecutado [illegible] usuario haga clic en dicho botón. Esto es excepcionalmente manejable por dos razones: una es que el código queda unido al objeto y la otra es que la ventana de código indica qué clase de suceso tiene que ocurrir para que se ejecute el código.

Además del suceso *Click*, hay otros sucesos asociados con un botón de órdenes. Observe en la figura siguiente, en la caja de procedimientos (**Proc:**), los sucesos que se pueden dar para el objeto particular *Orden1*.

[illegible]stinguir visualmente qué sucesos están asociados y cuales no, Visual Basic p[illegible]e en negrita los sucesos de la lista que vayamos asociando.



En la ventana de código también podemos ver, en la caja de objetos (**Object:**), la lista de todos los objetos pertenecientes a la forma sobre la que estamos trabajando. Para ver el procedimiento asociado a cualquiera de ellos, haga clic sobre su nombre.



Observe la lista, en ella están los nombres de la caja de texto (*Mensaje*), de los botones de órdenes (*Orden1, Orden2, Orden3*), y el nombre de la forma (*Form*). Quiere esto decir que también hay varios sucesos que pueden ocurrir con la forma. Por ejemplo *Load*; cuando una forma es utilizada por un programa por primera vez, éste considera automáticamente que ha ocurrido el suceso *Load* y ejecuta el procedimiento *Form_Load*. Además de los objetos mencionados hay otro objeto (*general*) que es añadido por Visual Basic. En la caja de código asociada con este objeto definiremos las variables y constantes comunes a todos los procedimientos.

Sigamos con nuestro ejemplo y pensemos la función que tiene que realizar cada uno de los controles. La caja de texto tiene que visualizar el texto que se envía cuando se pulsa el botón primero o el segundo. Quiere esto [illegible]que como tal objeto, no lleva código asociado.

Cuando el usuario haga clic sobre el botón *Haga clic aquí*, tiene que visualizarse en la caja de texto el mensaje "Bienvenido a Visual Basic". Por lo tanto el procedimiento asociado con este objeto es:

```
Sub Orden1_Click ()
  Mensaje.Text = "Bienvenido a Visual Basic"
End Sub
```

Haga doble clic en el botón *Haga clic aquí* y escriba el cuerpo de este procedimiento en la ventana de código presentada. Proceda de la misma forma para cada uno de los otros dos botones.

Cuando el usuario haga clic sobre el botón *?*, tiene que visualizarse en la caja de texto el mensaje "¿Contento?". Por lo tanto el procedimiento asociado con este objeto es:

```
Sub Orden2_Click ()
  Mensaje.Text = "¿Contento?"
End Sub
```

Cuando el usuario haga clic sobre el botón *Salir*, tiene que finalizar la aplicación. Por lo tanto el procedimiento asociado con este objeto es:

```
Sub Orden3_Click ()
  End
End Sub
```

Observe que al escribir el código se ha realizado una dentación o sangrado en las líneas (puede hacerlo con espacios en blanco o con tabulaciones). Esta dentación efectuada no es obligatoria pero es una buena práctica en programación, ya que de esta forma se consiguen programas más fáciles de entender. La dentación o sangrado se efectúa sobre un grupo de sentencias para indicar su pertenencia a otra sentencia.

Guardar la aplicación

Una vez finalizado el desarrollo de la aplicación se debe guardar en el disco para que pueda tener continuidad; por ejemplo si más adelante se quiere modificar. Para ello, ejecute la orden **Save Project** del menú **File** y asigne un nombre de fichero a la forma y al proyecto cuando le sean solicitados.

Si observamos el menú **File** nos encontramos además de con la orden **Save Project**, con tres órdenes más: **Save Project As...**, **Save File**, y **Save File As...**; también hay otra orden **Save Text** en el menú **Code**.

La orden **Save Project** guarda en el disco todos los ficheros asociados con la aplicación actual, en formato binario. La orden **Save Project As...** nos permite guardar la aplicación en el disco con otro nombre. Esto es útil, por ejemplo, cuando a partir de una aplicación existente realizamos otra.

La orden **Save File** guarda en el disco, en formato binario, la forma o módulo actualmente seleccionada/o y la orden **Save File As...** realiza la misma operación y además nos permite cambiar el nombre, lo cual es útil cuando la forma o el módulo ya existen.

La orden **Save Text** permite guardar en el disco, en texto normal, todos los ficheros que componen la aplicación. Esto es útil cuando queramos utilizar otro procesador de textos distinto al de Visual Basic, especialmente cuando el código que estamos escribiendo es demasiado largo y no está directamente ligado a la E/S. Esta es también la mejor forma de importar o de exportar texto escrito bajo QuickBasic, QBasic, o GW-Basic.

No es conveniente que utilice los nombres que Visual Basic asigna por defecto, porque pueden ser fácilmente sobrescritos al guardar aplicaciones posteriores.

Verificar la aplicación

Para ver como se ejecuta la aplicación y los resultados que produce, hay que ejecutar la orden **Start** del menú **Run** o pulsar **F5**.

Si durante la ejecución encuentra problemas o la solución no es satisfactoria y no es capaz de solucionarlos por sus propios medios, puede utilizar fundamentalmente las órdenes **Single Step (F8)**, **Procedure Step (Shift+F8)**, **Toggle Breakpoint (F9)**, todas ellas del menú **Run**, y la orden **Immediate Window** del menú **Window**.

La orden **Single Step** permite ejecutar cada procedimiento de la aplicación paso a paso. Esta modalidad se activa y se continúa pulsando **F8**. Si no quiere que los procedimientos y funciones se ejecuten línea a línea, sino de una sola vez, utilice las teclas **Shift+F8 (Procedure Step)**. Cada vez que el programa está detenido puede ver en la ventana inmediata los valores que tienen las variables.

La orden **Toggle Breakpoint** permite colocar un punto de parada (stop) en cualquier línea. Esto permite ejecutar la aplicación hasta el punto de parada sin pausa (**F5**), y ver en la ventana inmediata los valores de las variables en ese ins-

tante. Para poner o quitar una pausa, se coloca el cursor en el lugar donde va a tener o tiene lugar la pausa y se pulsa **F9**.

La orden **Immediate Window** abre una ventana donde podemos ejecutar de una forma inmediata cualquier sentencia. Por ejemplo,

```
?Mensaje.Text
```

El resultado será el contenido de esta variable.

Cuando se pulsa la tecla **F5**, la ejecución continua desde la última sentencia ejecutada en un procedimiento hasta finalizar ese procedimiento o hasta un punto de parada.

Crear un fichero ejecutable

Una vez que la aplicación tiene el aspecto deseable y que su ejecución transcurre satisfactoriamente, se puede crear un fichero ejecutable que permita ejecutar dicha aplicación fuera del entorno de Visual Basic.

Para guardar la aplicación como un fichero ejecutable, ejecute la orden **Make EXE File...** del menú **File**. Un fichero de este tipo requiere Windows y el fichero VBRUN100.DLL para poder ejecutarse.

Si el fichero **.EXE** se guarda en otro directorio distinto de VB (Visual Basic), tiene que guardar también en el mismo directorio una copia del fichero VRUN100.DLL. Este fichero se encuentra en el directorio VB.

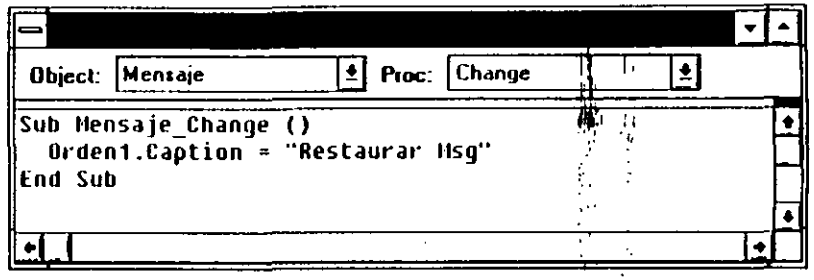
Cambio de propiedades en ejecución

Supongamos ahora que necesitamos realizar algunas modificaciones sobre nuestra anterior aplicación. Por ejemplo, nos hemos dado cuenta de que cuando se visualiza un mensaje, el usuario puede destruirlo escribiendo sin mas, sobre la caja de texto, y deseamos que esto no suceda.

Uno de los sucesos que se pueden dar sobre una caja de texto es que su contenido sea modificado al intentar escribir el usuario en la propia caja de texto. Cuando esto ocurre se activa el procedimiento correspondiente al suceso **Change** (cambiar) si es que se ha escrito para este control.

nuestra caja de texto se llama *Mensaje*, el procedimiento que se ejecuta es *Me...je_Change*. Entonces, lo que hay que hacer es interceptar el intento de

cambio del contenido de la caja de texto, añadiendo el código correspondiente a este procedimiento. Una posible solución se muestra en la figura siguiente.



Haciendo doble clic sobre la caja de texto se visualiza la ventana de código para este control. En ella se observa que uno de los sucesos que se pueden dar es **Change** como ya se ha dicho anteriormente. Nosotros hemos completado el procedimiento *Mensaje_Change* con la sentencia,

```
Orden1.Caption = "Restaurar Msg"
```

Esta sentencia lo que hace es cambiar el título *Haga clic aquí* del botón primero por el título *Restaurar Msg*. Esto sucederá siempre que el usuario intente cambiar el contenido de la caja de texto, y con ello se advierte al usuario que restaure el mensaje. Para que el botón primero, una vez restaurado el mensaje, vuelva a adquirir su título inicial, tendremos que modificar el procedimiento ligado al mismo de la forma siguiente:

```
Sub Orden1_Click ()
  Mensaje.Text = "Bienvenido a Visual Basic"
  Orden1.Caption = "Haga clic aquí"
End Sub
```

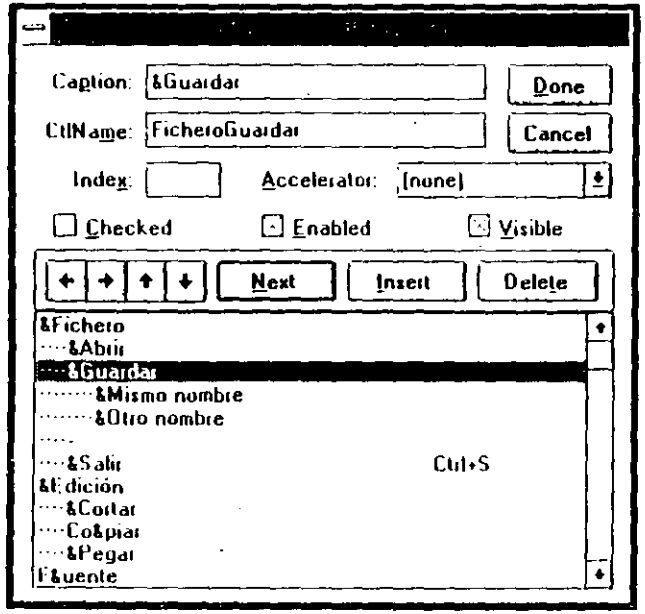
Ejecute el programa y observe como cada vez que intenta escribir sobre la caja de texto se ejecuta el procedimiento *Mensaje_Change*. Ahora pulse el botón *?* y sorpréndase de que también se ejecuta el procedimiento *Mensaje_Change*. Esto le enseña que cada vez que el contenido de una caja de texto se modifique, se ejecutará el suceso *Change* asociado a la misma.

1. *Abrir la ventana de diseño de menús* (**Menu Design Window**). Para ello, seleccione la forma para la que desea crear el menú y a continuación ejecute la orden **Menu Design Window** del menú **Window**.

2. *Introducir datos en la ventana.* En la caja de texto **Caption** se escribe el nombre del menú que se desea crear, el cual aparecerá en la barra de menús. Inserte un ampersand (&) antes de la letra que da acceso al menú. El usuario podrá seleccionar este menú pulsando la tecla **Alt** más la tecla que da acceso al menú, la cual aparece subrayada.

   En la caja de texto **CtlName** se escribe el nombre utilizado en el código para referirse al menú.

   A continuación, para crear el menú especificado en la caja **Caption**, haga clic en el botón **Next** o pulse **Enter**. Por ejemplo, en la figura siguiente *Fichero* y *Edición* son menús.



3. *Introducir las órdenes que componen el menú.* En la caja de texto **Caption** se escribe el nombre de la orden y en la caja de texto **CtlName** se escribe el nombre utilizado en el código para referirse a dicha orden. Por ejemplo, en la figura anterior *Abrir* es una orden.

Para diferenciar una orden de un menú del propio menú, hay que sangrar o dentar la orden. Para sangrar una orden selecciónela y haga clic en el botón flecha hacia la derecha (→).

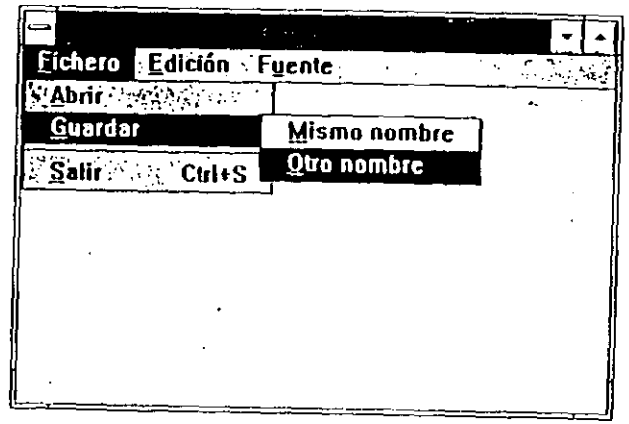
Después de una orden, se puede añadir otra orden o bien, otro menú.

4. *Crear un submenú.* Puede crear un submenú sangrando una orden debajo de otra orden. Por ejemplo, en la figura anterior *Guardar* es un submenú.

5. *Añadir una línea de separación.* Utilizando líneas de separación puede agrupar las órdenes en función de su actividad. Para crear una línea de separación escriba un único guión (-) en la caja **Caption** de la ventana de diseño de menús. Tiene que especificar también un nombre cualquiera (**CtlName**). Por ejemplo, en la figura anterior se ha creado una línea de separación antes de la orden *Salir*.

6. *Cerrar la ventana de diseño de menús.* Una vez que haya finalizado el diseño pulse la tecla **Done** y observe como en la barra de menús de la forma aparecen los menús diseñados.

Guarde la aplicación y ejecutándola (F5) o no, haga clic sobre cualquier menú y se sorprenderá al ver como se despliega. Lógicamente, las órdenes que componen cada menú no estarán operativas hasta no unir el código correspondiente.



Para definir como debe responder cada orden de un menú al suceso **Click**, hay que escribir un procedimiento para cada una de ellas. Las órdenes de un menú sólo responden al suceso **Click**. Por ejemplo, a continuación se muestra el pro-

cedimiento para la orden *Salir* del menú *Fichero* que se visualiza en la figura anterior, cuyo nombre de control (**CtlName**) es *FicheroSalir*.

```
Sub FicheroSalir_Click ()
   End
End Sub
```

Para crear un procedimiento para una orden de un menú, haga clic sobre el menú y a continuación haga clic sobre la orden. Observe que se visualiza el esquema del procedimiento, lo que le permite escribir el mismo.

Otra forma de realizar este proceso es la siguiente:

1. Para abrir la ventana de código ejecute la orden **View Code** del menú **Code** o pulse **F7**.
2. Seleccione el nombre del control en la caja **Object**.
3. Escriba el código entre **Sub** y **End Sub**.

Propiedades de un menú

Las propiedades que pueden seleccionarse en la ventana de diseño de menús son **Accelerator**, **Checked**, **Enabled**, **Visible** e **Index**.

La propiedad **Accelerator** permite definir una tecla o combinación de teclas para ejecutar una orden. Las combinaciones válidas son **Ctrl+A** a **Ctrl+Z**, **F1** a **F12**, **Ctrl+F1** a **Ctrl+F12**, **Shift+F1** a **Shift+F12** y **Ctrl+Shift+F1** a **Ctrl+Shift+F12**. Por ejemplo, abra la ventana de diseño de menús, seleccione la orden *Salir* y elija para la propiedad **Accelerator** la combinación **Ctrl+S**. Cierre la ventana de diseño de menús y ejecute la aplicación. Compruebe como al pulsar **Ctrl+S** se ejecuta la orden *Salir*.

La propiedad **Checked** es útil para indicar si una orden está activa o no lo está. Cuando se especifica esta propiedad aparece una marca ✓ a la izquierda del elemento del menú. Por ejemplo, puede utilizar esta propiedad en el menú *Fuente* de la figura anterior para especificar el tipo de letra que está activo.

La propiedad **Enabled** es útil para desactivar una orden en un momento en el cual no tiene sentido que esté activa. Por ejemplo, si estamos trabajando con un editor y no tenemos seleccionado un bloque de texto, no tiene sentido que la orden *Cortar* del menú *Edición* esté activa. Cuando se hace clic en una orden de un menú será ejecutada si su propiedad **Enabled** está señalada. Si la propiedad **Enabled** no está señalada, la orden se muestra en tono gris y no puede ser ejecutada.

La propiedad **Visible** es útil cuando en tiempo de ejecución se desea ocultar una orden. Cuando esta propiedad no está señalada, la orden no aparece y por lo tanto no puede ejecutarse.

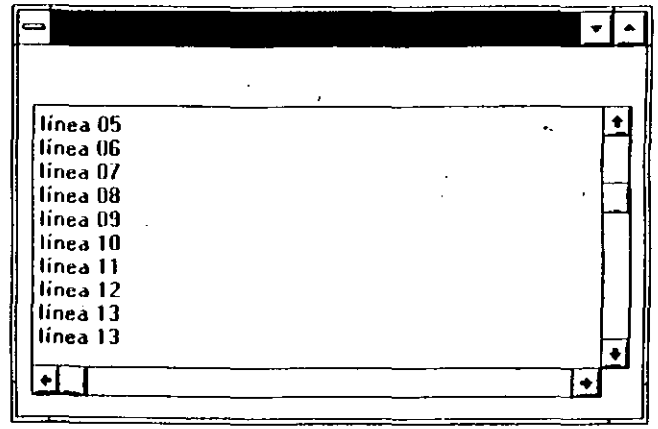
La propiedad **Index** permite que un conjunto de órdenes sean agrupadas en un array de controles. Cuando un usuario selecciona una orden de un array de controles, Visual Basic utiliza el valor **Index** para identificar qué orden del array fue seleccionada.

CAJA DE TEXTO CON MÚLTIPLES LÍNEAS

Un editor de textos parece a simple vista una aplicación muy complicada, pero ahora veremos que no es así. Visual Basic tiene mecanismos de entrada que hacen sumamente sencillas aplicaciones como ésta.

Utilizando una definición sencilla, un editor de textos es una caja de texto con múltiples líneas, y Visual Basic soporta este tipo de cajas.

El aspecto y el comportamiento de una caja de texto está altamente influenciado por dos propiedades, **MultiLine** y **ScrollBars**.



Para crear el editor que vemos en la figura los pasos son muy sencillos:

1. Abra una nueva forma.

2. Dibuje sobre ella una caja de texto y ajústela al tamaño que desee. Seleccione la propiedad **Text** y asígnela un valor nulo.

3. Seleccione la propiedad **MultiLine** y asígnela el valor **True**. Esto permite escribir en la caja varias líneas de texto. Por defecto esta propiedad tiene el valor **False** lo que indica que la caja de texto sólo puede contener una línea.

4. Seleccione la propiedad **ScrollBars** y asígnela el valor **3 - Both**. De esta forma, la caja de texto queda dotada con barras de desplazamiento horizontal y vertical. Por defecto esta propiedad tiene valor **0** lo cual indica que no hay barras de desplazamiento. Un valor **1** añade a la caja de texto solamente la barra de desplazamiento horizontal y un valor **2** añade a la caja de texto solamente la barra de desplazamiento horizontal.

Guarde la aplicación y ejecútela. Pruebe a escribir texto, actúe sobre las barras de desplazamiento, modifique el texto, inserte texto, y seleccione, borre y mueva bloques de texto. Como podrá comprobar todas estas operaciones están implícitas sin escribir nada de código.

Cuando una línea de texto es más larga de lo que puede ser, automáticamente se produce un salto a la línea siguiente. Esta longitud no coincide con el ancho de la caja de texto.

También se puede añadir texto en tiempo de ejecución. Por ejemplo, escriba el procedimiento *Form_Load* que se indica a continuación:

```
Sub Form_Load ()
  NLS = Chr$(13) + Chr$(10) 'Carácter nueva línea
  Text1.Text = "Línea 01" + NLS + "Línea 02"
End Sub
```

Este procedimiento hace que se visualice en la caja dos líneas de texto, "Línea 01" y "Línea 02".

Para hacer más operativo nuestro editor, vamos a añadirle las órdenes de *Cortar*, *Copiar* y *Pegar*. Estas órdenes nos permitirán seleccionar un texto y desplazarlo dentro del mismo documento o llevarlo a otro documento.

TRABAJAR CON TEXTO SELECCIONADO

En cualquier editor de texto el usuario puede seleccionar texto utilizando el ratón o el teclado. Con el ratón, apunte y arrastre con el botón izquierdo pulsado y con el teclado, sitúe el cursor y manteniendo pulsada la tecla **Shift** (mayúsculas) desplace el cursor utilizando las teclas de desplazamiento del mismo. Visual Basic da automáticamente esta capacidad a las cajas de texto. Además de esto, las cajas de texto tienen una serie de propiedades que permiten trabajar con el texto seleccionado a través del portapapeles (**Clipboard**). Estas propiedades son, **SelStart**, **SelLenght** y **SelText** y se refieren al bloque de texto seleccionado.

La propiedad **SelStart** es un entero que especifica la posición de comienzo del bloque de texto seleccionado. Si no hay texto seleccionado, entonces esta propiedad hace referencia a la posición de inserción. Un valor cero especifica la posición justamente antes del primer carácter de la caja de texto y un valor igual a la longitud del texto especifica la posición justamente después del último carácter de la caja de texto.

La propiedad **SelLenght** es un entero que especifica el número de caracteres seleccionados.

La propiedad **SelText** es una cadena de caracteres que contiene el texto seleccionado. Si no hay texto seleccionado, su valor es nulo.

UTILIZANDO EL PORTAPAPELES

El portapapeles es un objeto que no tiene propiedades ni sucesos, pero sí tiene una serie de métodos (procedimientos) que permiten transferir datos a y desde su entorno. Los métodos más útiles son **SetText** y **GetText**.

El método **SetText** copia el texto seleccionado en el portapapeles destruyendo cualquier otro texto que hubiera en el mismo y el método **GetText** recupera el texto almacenado en el portapapeles. La sintaxis para cada uno de estos métodos es la siguiente:

Clipboard.SetText *origen*

destino = **Clipboard.GetText()**

Ejemplos:

```
Clipboard.SetText Text1.SelText
```

Esta sentencia guarda en el portapapeles el texto seleccionado en la caja de texto *Text1*.

```
Text1.SelText = Clipboard.GetText()
```

Esta otra sentencia coge el contenido del portapapeles y lo inserta en la caja de texto *Text1* a partir de la posición del cursor.

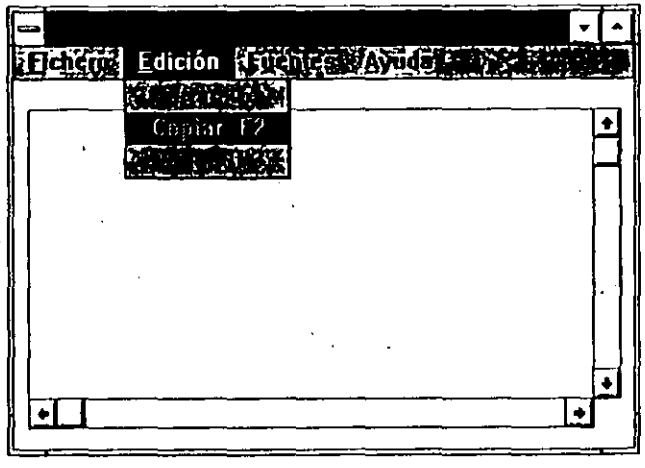
Otro método útil que permite borrar todo el contenido del portapapeles es **Clear**. La sintaxis es la siguiente:

Clipboard.Clear

DESARROLLO DE UN EDITOR

Nuestra próxima aplicación va a consistir en el desarrollo de un editor de textos. Este editor, aunque sus prestaciones son muy limitadas, va a servir para poner en práctica los menús, las cajas de texto con múltiples líneas y la utilización del portapapeles, denominado **Clipboard** en Visual Basic.

Para comenzar vamos a diseñar la forma y los controles que se muestran en la figura siguiente.



En primer lugar cree una nueva forma y asígnela el nombre *Editor*.

Añada a la forma una caja de texto, ajuste su tamaño para formar el área de escritura del editor y asígnela las siguientes propiedades:

CtlName	Text1
M `ne`	True
Sc `s`	3 - (Ambas)
Text	(ninguno)

A continuación cree los menús *Fichero*, *Edición*, *Fuentes* y *Ayuda*. Para este planteamiento inicial, suponga que el menú *Fichero* está formado sólo por la orden *Salir*, el menú *Edición* por las órdenes *Cortar*, *Copiar* y *Pegar*, el menú *Fuentes* por las órdenes *Courier*, *Helv* y *Times* y el menú *Ayuda* por la orden *Acerca de*. Las propiedades para cada uno de los menús se resumen en la tabla siguiente.

Caption	CtlName	Accelerator	Enabled
Fichero	MenúFichero	ninguno (none)	Si
Salir	FicheroSalir	Ctrl+S	Si
Edición	MenúEdición	ninguno	Si
Cortar	EdiciónCortar	F1	No
Copiar	EdiciónCopiar	F2	No
Pegar	EdiciónPegar	F3	No
Fuentes	MenúFuentes	ninguno	Si
Courier	Fuente (Index: 0)	ninguno	Si
Helv	Fuente (Index: 1)	ninguno	Si
Symbol	Fuente (Index: 2)	ninguno	Si
Ayuda	MenúAyuda	ninguno	Si
Acerca de	AyudaAcercaDe	ninguno	Si

Para el resto de las propiedades se asume el valor por defecto.

A continuación escribimos el código correspondiente para cada una de las órdenes. Para visualizar la ventana de código para una determinada orden, haga clic en el menú y después en la orden. A continuación escriba el código entre **Sub** y **End Sub**.

El menú *Fichero* sólo tiene la orden *Salir*. Cuando esta orden se ejecute la aplicación finalizará. Por lo tanto el procedimiento es el siguiente:

```
Sub FicheroSalir_Click ()
  End
End Sub
```

Más adelante, cuando lleguemos a manipular ficheros, añadiremos a este menú órdenes como *Abrir Fichero* y *Guardar Fichero*.

El menú *Edición* tiene tres órdenes *Cortar*, *Copiar* y *Pegar* que inicialmente no pueden ser ejecutadas (su propiedad **Enabled** no está señalada). Las órdenes *Cortar* y *Copiar* estarán activas cuando haya seleccionado un bloque de texto y la orden *Pegar* estará activa cuando haya texto en el portapapeles. Con el fin de ir explicando nuevos conceptos, supongamos que la selección de texto para *Cortar* y *Copiar* la realizamos utilizando el teclado, esto es, pulsando **Shift**+*tecla* (donde *tecla* es, ←, ↑, →, ↓ cuyos códigos 25H, 26H, 27H y 28H se pueden ver en el fichero CONSTANT.TXT de Visual Basic).

Según lo expuesto, la condición para activar las órdenes *Cortar* y *Copiar* es que el usuario haya seleccionado un bloque de texto, lo que puede saberse interceptando las teclas que pulsa (recuerde que cuando se pulsa una tecla se dan tres sucesos **KeyPress**, **KeyDown** y **KeyUp**; ver Capítulo 4). Para activar una orden hay que poner la propiedad **Enabled** de la misma a un valor -1 y para desactivarla a un valor 0. Todo esto puede especificarse de la forma siguiente:

```
If (Shift = 1) And (KeyCode = &H25 Or KeyCode = &H26
    Or KeyCode = &H27 Or KeyCode = &H28) Then
  EdiciónCortar.Enabled = -1
  EdiciónCopiar.Enabled = -1
End If
```

La condición para desactivar las órdenes *Cortar* y *Pegar* es que no haya texto seleccionado, lo que puede saberse por la propiedad **SelLength** (longitud del texto seleccionado) de la caja de texto.

```
If Text1.SelLength = 0 Then
  EdiciónCortar.Enabled = 0
  EdiciónCopiar.Enabled = 0
End If
```

La orden *Pegar* estará activa cuando haya texto en el portapapeles y no lo estará cuando el portapapeles esté vacío. Recuerde que el método **GetText()** del objeto **Clipboard** devuelve como resultado una cadena de caracteres igual a su contenido. Si la longitud de esta cadena es 0, el portapapeles está vacío. Esto puede averiguarse de la forma siguiente:


```
If Len(Clipboard.GetText()) = 0 Then
  EdiciónPegar.Enabled = 0
Else
  EdiciónPegar.Enabled = -1
End If
```

La función **Len** da como resultado la longitud o número de caracteres de una cadena.

A continuación agrupamos los segmentos de código anteriores en un procedimiento que se ejecute cada vez que el usuario pulse una tecla. Vistas las teclas que utilizamos para seleccionar un bloque de texto, el procedimiento que escribamos tiene que ser conducido por el suceso **KeyDown** o **KeyUp**. El suceso **KeyDown** ocurre cada vez que el usuario pulsa una tecla y el suceso **KeyUp** ocurre cada vez que el usuario suelta la tecla pulsada.

```
Sub Text1_KeyUp (KeyCode As Integer, Shift As Integer)
  If Shift = 1 And (KeyCode = &H25 Or KeyCode = &H26
      Or KeyCode = &H27 Or KeyCode = &H28) Then
    EdiciónCortar.Enabled = SI
    EdiciónCopiar.Enabled = SI
  End If
  If Text1.SelLength = 0 Then
    EdiciónCortar.Enabled = NO
    EdiciónCopiar.Enabled = NO
  End If
  If Len(Clipboard.GetText()) = 0 Then
    EdiciónPegar.Enabled = NO
  Else
    EdiciónPegar.Enabled = SI
  End If
End Sub
```

Las constantes SI y NO las definiremos a nivel de la forma con los valores -1 y 0 respectivamente.

```
Const SI = -1
Const NO = 0
```

Una vez seleccionado un bloque de texto podemos copiarlo o cortarlo. El bloque resultante de esta operación se puede almacenar en una cadena de caracteres cualquiera, o en el portapapeles. La ventaja de utilizar el portapapeles, es que éste permite intercambiar información entre aplicaciones que lo utilicen. Esto es,

podemos llevar bloques de información de una aplicación a otra utilizando las órdenes de *Cortar*, *Copiar* y *Pegar*.

Cuando el usuario haga clic sobre la orden *Cortar* se ejecutará el procedimiento asociado con ella, el cual, según se muestra a continuación, tiene que almacenar el texto seleccionado en el portapapeles y después borrarlo de la ventana.

```
Sub EdiciónCortar_Click ()
  Clipboard.SetText Text1.SelText
  Text1.SelText = ""
End Sub
```

La orden *Copiar* se diferencia de la orden *Cortar* en que no borra el texto seleccionado.

```
Sub EdiciónCopiar_Click ()
  Clipboard.SetText Text1.SelText
End Sub
```

Cuando el usuario haga clic sobre la orden *Pegar* se ejecutará el procedimiento *EdiciónPegar_Click*, el cual, según se muestra a continuación, colocará el texto del portapapeles sobre el texto seleccionado o en su defecto, a partir de la posición del cursor.

```
Sub EdiciónPegar_Click ()
  Text1.SelText = Clipboard.GetText()
End Sub
```

Para asegurar que el portapapeles esté vacío cuando arranquemos la aplicación, escribiremos el procedimiento siguiente:

```
Sub Form_Load ()
  Clipboard.Clear
End Sub
```

Cuando los menús sean largos, un detalle de gusto a tener en cuenta es no presentar casi todas las opciones desactivadas (en tono gris). Una solución mejor es eliminar las órdenes que menos se utilicen del menú y reemplazarlas cuando sean necesarias. Esto se consigue poniendo la propiedad **Visible** de la orden, a valor −1 (verdadero) o a valor 0 (falso).

A continuación vamos a desarrollar el código para el menú *Fuentes*. Una fuente es un tipo de letra. El número de fuentes disponibles en Visual Basic varía en función de la configuración de su sistema. En nuestro editor sólo vamos a incluir tres tipos: **Courier**, **Helvética** y **Symbol**. Para cambiar el editor de textos a una de estas fuentes simplemente tenemos que modificar la propiedad **FontName** de la caja de texto.

En una caja de texto solamente puede utilizarse una fuente, lo cual significa que cuando se modifique el tipo de letra para la caja de texto, cambiará todo el texto a ese tipo de letra.

Si usted quiere utilizar diferentes tipos de letras en el mismo documento, tiene que utilizar el método **Print** que se aplica a formas y a cajas de imágenes. Esta sería la solución para crear un editor que admita diferentes tipos de letras y que pueda manipular un texto mayor de 64K, valor límite para una caja de texto.

Las órdenes del menú *Fuentes* forman un array de controles denominado *Fuente* (*Fuente(0)* = **Courier**, *Fuente(1)* = **Helvética** y *Fuente(2)* = **Symbol**). Esto evita tener que escribir un procedimiento para cada tipo de letra.

Cuando el usuario haga clic en una orden del menú *Fuentes*, el procedimiento *Fuente_Click* será invocado para su ejecución y recibirá como valor para **Index** el índice asignado cuando creó este menú (0 = **Courier**, 1 = **Helvética** y 2 = **Symbol**). Por lo tanto, el cuerpo del procedimiento lo único que debe hacer es verificar qué valor se pasa y asignar en función de este valor la cadena de caracteres correspondiente al nombre de la fuente.

```
Sub Fuente_Click (Index As Integer)
  Select Case Index
    Case 0
      Text1.FontName = "Courier"
    Case 1
      Text1.FontName = "Helv"
    Case 2
      Text1.FontName = "Symbol"
  End Select
End Sub
```

A continuación vamos a señalar (propiedad **Checked**) la fuente que está activa. Visual Basic utiliza por defecto el tipo de letra "Helv". Quiere esto decir que cuando el usuario arranque la aplicación, esta es la fuente que debe aparecer señalada. Esto se puede hacer incluyendo en el procedimiento *Form_Load* la sentencia *Fuente(1).Checked = -1* o abriendo la ventana de diseño de menús, eligiendo la orden "Helv" y señalando la propiedad **Checked**.

Durante la ejecución, cuando el usuario seleccione otro tipo de letra, hay que quitar la señal de donde esté y ponerla en el tipo de letra seleccionado para reali-

zar esta operación, defina a nivel de la forma la variable *Ind* para guardar el índice del tipo de letra actual.

```
Dim Ind As Integer
```

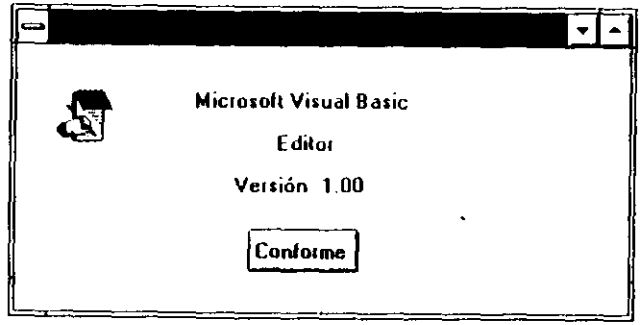
Inicialice esta variable en el procedimiento *Form_Load* al valor del tipo de letra por defecto.

```
Ind = 1    'Tipo de letra por defecto Helv
```

Por último, modifique el procedimiento *Fuente_Click* como se indica a continuación.

```
Sub Fuente_Click (Index As Integer)
  Fuente(Ind).Checked = NO  'quita señal actual
  Select Case Index
    Case 0
      Text1.FontName = "Courier"
    Case 1
      Text1.FontName = "Helv"
    Case 2
      Text1.FontName = "Symbol"
  End Select
  Fuente(Index).Checked = SI 'pone señal
  Ind = Index  'guarda índice de tipo de letra actual
End Sub
```

A continuación pasamos al diseño del menú *Ayuda*. Este menú está compuesto por una sola orden, *Acerca de*, y lo que queremos es que cuando esta orden se ejecute, aparezca la ventana siguiente:



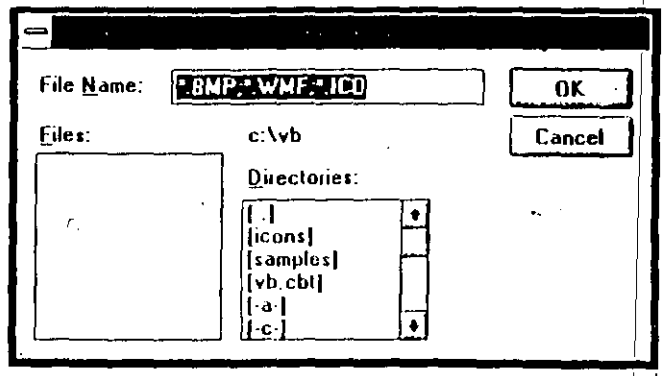
Para crear esta ventana siga los pasos que se describen a continuación:

1. Cree una nueva forma y ajuste su tamaño. Asígnela el título (**Caption**) *Acerca del Editor* y el nombre (**FormName**) *AcercaDe*.

2. Dibuje los controles siguientes:

	CtlName	Caption
Etiqueta 1	Label1	Microsoft Visual Basic
Etiqueta 2	Label2	Editor
Etiqueta 3	Label3	Versión 1.00
Botón	Conforme	Conforme

3. Añada una imagen (en este caso un icono) a la forma que simbolice su función. Para ello, añada una caja de imagen a la forma (primera fila, segunda columna, del panel de utilidades). Seleccione en la barra de propiedades la propiedad **Picture** de la caja de imagen, y haga clic en el botón [...] situado a la derecha. Visual Basic visualiza una caja de diálogo como la siguiente:



Si no está situado en el directorio **vb**, sitúese haciendo doble clic sobre [..] las veces que sean necesarias. Elija el directorio **icons** y dentro de éste el directorio **writing**. Cargue el icono **note17.ico** (haga doble clic sobre ese nombre o selecciónelo y pulse el botón **OK**).

Ahora tenemos creada la forma. El siguiente paso es escribir el procedimiento correspondiente a la orden *Acerca de*. Este es el siguiente:

```
Sub AyudaAcercaDe_Click ()
  AcercaDe.Show 1  'l = Conformar
End Sub
```

El método **Show** permite cargar y visualizar cualquier forma en la aplicación. Si la forma ya está cargada pero no visualizada (está oculta - **Hide**), **Show** la visualiza. Su sintaxis es la siguiente:

[*forma.*]**Show** [*modo%*]

El argumento *modo* es un entero de valor 0 o 1, por defecto es 0. Un valor 1 obliga a cerrar la forma visualizada para poder continuar (descargarla u ocultarla).

Cuando Visual Basic carga una forma, además de ponerla sus propiedades ejecuta el procedimiento *Form_Load* asociado con la misma.

Para cerrar esta forma hemos dispuesto el botón *Conforme*. Por lo tanto, el código asociado con este botón es el siguiente:

```
Sub Conforme_Click ()
  AcercaDe.Hide
End Sub
```

El método **Hide** oculta (no descarga) la forma especificada de la aplicación. La ventaja de ocultar una forma en lugar de descargarla, es que los datos y propiedades ligados a la misma no se pierden. Su sintaxis es la siguiente:

[*forma.*]**Hide**

El botón *Conforme* responde a un clic, a la tecla **Enter** o a la tecla **espacio**. Para hacer que responda a cualquier otra tecla, escriba el siguiente procedimiento:

```
Sub Conforme_KeyDown (KeyCode As Integer, Shift As Integer)
  Conforme_Click
End Sub
```

El procedimiento *Conforme_KeyDown* está asociado con el objeto *Conforme* y se ejecuta para el suceso **KeyDown** que se da cada vez que el usuario pulsa una tecla

APLICACIONES CON MÚLTIPLES FORMAS

Cualquier aplicación Visual Basic puede ser extendida a más de una forma, cada una de las cuales tiene su propio aspecto y propósitos. Un ejemplo claro es la aplicación anterior, en la que se hizo prácticamente todo el diseño sobre una forma pero al final resultó necesario añadir una forma más para dar información acerca de la aplicación. Está claro que la utilización de varias formas incrementa la funcionalidad de la aplicación.

Disponer de más de una forma es particularmente útil cuando se necesita crear cajas de diálogo, como veremos más tarde en este capítulo y en el próximo capítulo.

Métodos y sentencias para manipular formas

Para manipular formas Visual Basic dispone de los métodos **Show** y **Hide**, y de las sentencias **Load** y **Unload**.

Los métodos **Show** y **Hide** ya fueron expuestos anteriormente en este mismo capítulo y las sentencias **Load** y **Unload** las comentamos a continuación.

La sentencia **Load** permite cargar una forma en la memoria pero no la visualiza. Para tener acceso a los controles y propiedades de una forma, ésta debe estar cargada en memoria. Su sintaxis es:

Load *forma*

La sentencia **Unload** descarga (cierra) una forma de la memoria independientemente de que se esté o no visualizando.

Cuando se trabaja con múltiples formas, puede ser necesario referirse desde el código ligado a una de ellas, a los controles y propiedades de otra forma. O también, puede ser necesario escribir un procedimiento en un módulo para que lo puedan utilizar más de una forma. En ambos casos, es necesario calificar las referencias a las formas.

Para referirse a una propiedad de otra forma, se utiliza la sintaxis,

forma.propiedad

Para referirse a una propiedad de un control de otra forma, se utiliza la sintaxis,

forma.control.propiedad

Cuando la aplicación tiene una sola forma, Visual Basic asume el nombre de la forma actual, excepto en los módulos, donde hay que especificarlo explícitamente.

ASOCIAR UN ICONO A LA APLICACIÓN

Después de finalizar una aplicación, puede crear un fichero ejecutable bajo Windows (vea crear un fichero .EXE en el Capítulo 2). Una vez creado el fichero ejecutable, puede añadir esta aplicación en la ventana de "Aplicaciones Windows" del "Administrador de Programas" de Windows. Para hacerlo siga los siguientes pasos:

1. Seleccione la ventana "Aplicaciones Windows".
2. Ejecute la orden **Nuevo...** del menú **Archivo**.
3. En el cuadro de diálogo elija "Elemento de programa" y pulse **Aceptar**.
4. Edite las propiedades del elemento de programa,

Descripción:	Editor
Línea de comando:	Editor
Directorio de trabajo:	c:\vb

5. Pulse **Aceptar**.

Observará que se ha añadido a la ventana "Aplicaciones Windows" el icono correspondiente a nuestra aplicación. Este icono puede ser el icono por defecto de Visual Basic, o un icono asociado durante el diseño de la aplicación.

Para asociar un icono a la aplicación, haga clic sobre la forma, seleccione la propiedad **Icon** y proceda igual que se explicó en el apartado anterior para añadir una imagen a la forma.

EL FICHERO CONSTANT.TXT

Visual Basic incluye un fichero denominado CONSTANT.TXT que define por defecto una serie de constantes relativas a colores, códigos del teclado, propiedades y sucesos. Por ejemplo,

```
Global Const TRUE = -1
Global Const FALSE = 0
```


Si quiere utilizar las definiciones de este fichero puede cargarlo o fusionarlo en el módulo global de la aplicación. Para cargar CONSTANT.TXT en éste módulo.

1. Seleccione el módulo **global.bas** en la ventana de la aplicación.
2. Ejecute la orden **Load Text** del menú **Code**.
3. Seleccione el fichero CONSTANT.TXT.
4. Pulse el botón **Replace** o **Merge**.

Si usted sólo necesita algunas definiciones del fichero CONSTANT.TXT ejecute sólo los puntos 2 y 3 y pulse el botón **New**. A continuación seleccione del módulo creado las definiciones que necesite y páselas, a través del portapapeles de Visual Basic, al módulo **global.bas** (copiar y pegar). Esto reduce la cantidad de memoria necesaria para la aplicación. Después elimine el módulo que creó ejecutando la orden **Remove File** del menú **File**.

MODIFICAR UN MENÚ EN TIEMPO DE EJECUCIÓN

Una propiedad de un control es especificada por defecto o explícitamente durante el diseño y puede ser modificada durante la ejecución de la aplicación, sin mas que asignarle un valor de su tipo. Por ejemplo,

```
EdiciónCortar.Enabled = -1
Text1.FontName = "Courier"
```

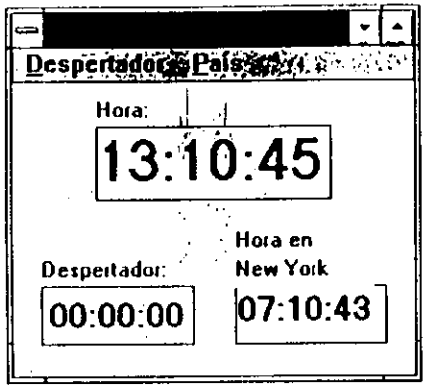
Igualmente una orden de un menú, puede ser cambiada, borrada o añadida durante la ejecución. Veamos estas afirmaciones con un ejemplo.

DESARROLLO DE UN RELOJ DESPERTADOR

Vamos a diseñar un reloj despertador digital como el que se muestra en la figura siguiente. El reloj tiene una pantalla para visualizar la hora y una caja denominada *Despertador* donde el usuario puede escribir la hora a la que quiere ser avisado. Para activar o desactivar el despertador, el usuario dispone de un menú denominado también *Despertador*.

La pantalla para visualizar la hora será una etiqueta con el fin de que el usuario no pueda modificarla. Para que la hora varíe segundo a segundo, el título de la etiqueta a la que hemos hecho referencia, que representa la hora, debe cambiarse a intervalos iguales o inferiores a un segundo. Para realizar esto Visual Basic dispone de un control denominado temporizador.

Hay un segundo menú denominado *País* que permitirá al usuario añadir países al propio menú, para después haciendo clic sobre cualquiera de ellos, obtener en una tercera caja, formada por una etiqueta, la hora actual en ese país.



Temporizador

Un temporizador es un control de Visual Basic que responde a intervalos regulares de tiempo. Esto quiere decir que en el procedimiento asociado con el mismo, especificaremos las acciones que deseemos se ejecuten cada vez que transcurra un intervalo de tiempo.

Cada temporizador tiene una propiedad **Interval** que especifica el intervalo de tiempo en milisegundos que tiene que transcurrir para que su procedimiento asociado se ejecute independientemente del usuario. El valor de la propiedad **Interval** puede oscilar entre 0 y 64767 (0 a 64.8 segundos).

El sistema genera 18 tics de reloj por segundo, por ello aunque el valor de la propiedad **Interval** se exprese en milisegundos, la precisión no puede ser mayor de 1000/18 milésimas de segundo.

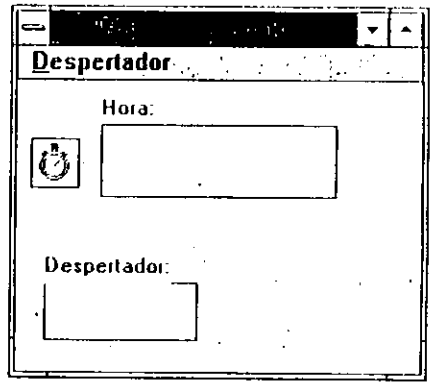
Una utilidad típica de este control es verificar, de alguna forma, la hora del sistema para ver si es el momento de ejecutar alguna tarea.

Diseño de la forma y de los controles

Inicie una nueva aplicación y asigne a la forma por defecto el título (**Caption**) *Reloj Despertador*; ahora estamos listos para añadir los controles. Lo primero que vamos a hacer es colocar la pantalla del reloj: haga un doble clic en la utilidad etiqueta del panel de utilidades y coloque la etiqueta en el lugar que se indica en la figura; ajuste su tamaño y asígnela las siguientes propiedades:

BorderStyle:	1	Estilo del borde
Caption:	(nada)	Título
CtlName:	Hora	Nombre del control
BackColor:	&H0000FFFF&	Color de fondo
ForeColor:	&H00000000&	Color del primer plano
FontSize:	24	Tamaño del texto

Añada una etiqueta encima del borde de la anterior, asígnela el título *Hora:*, ajuste el tamaño y sitúela como indica la figura.



Colocamos ahora un temporizador, como se ve en la figura, por ejemplo a la izquierda de la etiqueta *Hora:*; haga un doble clic en la utilidad temporizador del panel de utilidades y sitúe el temporizador en el lugar indicado. Ponga el valor de la propiedad **Interval** a valor 1000 (intervalos de 1 segundo).

A continuación añadimos una caja de texto que va a dar lugar al despertador; haga un doble clic en la utilidad caja de texto del panel de utilidades y coloque la caja en el lugar que se indica en la figura; ajuste su tamaño y asígnela las siguientes propiedades:

CtlName:	Despertador	Nombre del control
BackColor:	&H0000FFFF&	Color de fondo
ForeColor:	&H00000000&	Color del primer plano
FontSize:	13.5	Tamaño del texto

Añada una etiqueta encima del borde de la caja de texto, asígnela el título *Despertador:*, ajuste el tamaño y sitúela como indica la figura.

Para activar o desactivar el despertador vamos a añadir un menú. Abra la ventana de diseño de menús y diseñe el menú *Despertador* con las órdenes *Despertador No* y *Cerrar* (deje las propiedades especificadas por defecto).

Caption:	Despertador No	**CtlName:**	DespertadorSiNo
Caption:	Cerrar	**CtlName:**	Cerrar

Es una buena idea, guardar ahora el trabajo realizado: ejecute la orden **Save Project** y asigne a la forma el nombre *reloj.frm* y a la aplicación (proyecto) el nombre *reloj.mak*.

Unir el código a los controles y a la forma

Tenemos ya los controles necesarios para nuestro reloj. Ahora uniremos a ellos el código correspondiente para un correcto funcionamiento del reloj.

Empecemos por el despertador: nuestra intención es que el usuario escriba en esta caja la hora a la que quiere ser avisado. Por lo tanto los cáracteres que él pueda escribir los restringiremos a los dígitos "0" a "9" y los ":". Para realizar el procedimiento correspondiente, haga un doble clic sobre la caja de texto despertador, seleccione el suceso **KeyPress** y escriba.

```
Sub Despertador_KeyPress (KeyAscii As Integer)
  Dim Car As String * 1
  Car = Chr$(KeyAscii)
  If (Car < "0" Or Car > "9") And Car <> ":" Then
    Beep            'pitido
    KeyAscii = 0    'borrar carácter
  End If
End Sub
```

Este procedimiento recibe como argumento el código de la tecla pulsada y lo almacena en el parámetro **KeyAscii**. La función **Chr$** convierte este código a su correspondiente carácter ASCII y lo almacena en la variable *Car*. A continuación se verifica si el carácter no es uno de los permitidos, en cuyo caso se emite un pitido y se borra el carácter, esto es, se cambia el carácter tecleado por el carácter nulo (código 0).

El paso siguiente es presentar la hora a través del control *Hora*. Para visualizar la hora Visual Basic tiene la función **Time$** la cual devuelve una cadena de 8

caracteres de la forma hh:mm:ss. Entonces, para presentar la hora bastaría con escribir una sentencia como,

```
Hora.Caption = Time$
```

La cuestión es que si esta sentencia se ejecuta una sola vez, la hora será la presentada y no variará. Es preciso por lo tanto, ejecutarla a intervalos de un segundo o menos. Para conseguir esto, haga que esta sentencia sea parte del cuerpo del procedimiento del temporizador, el cual ha sido programado para que dicho procedimiento se ejecute cada segundo. Para ello, haga un doble clic sobre el temporizador y escriba:

```
Sub Timer1_Timer ()
  Hora.Caption = Time$
End Sub
```

Por [illegible] y algo más, supongamos que el usuario escribe una hora en el despertador. Para que se produzcan pitidos de aviso cuando se alcance esa hora, tienen que darse dos condiciones: una que el despertador esté activado y otra que la hora actual sea igual o mayor que la hora especificada en el despertador. La condición mayor es para que el pitido continúe hasta que el usuario desactive el despertador.

Defina una variable *DespertadorSi* a nivel de la forma, y asígnela un valor inicial cero. Un valor **0** (falso - **False**) significa despertador desactivado y un valor **–1** (verdad - **True**) significa despertador activado.

```
Dim DespertadorSi As Integer
```

Para inicializar las variables el procedimiento adecuado es *Form_Load* ya que es el primero que se ejecuta cuando se arranca la aplicación.

```
Sub Form_Load ()
  DespertadorSi = False
  Despertador.Text = "hh:mm:ss"
End Sub
```

El valor **False** está definido en el fichero CONSTANT.TXT. Abra este fichero, seleccione las definiciones que le interesen y colóquelas en el fichero **global.bas**, que a continuación guardaremos con el nombre *reloj.bas*.

De acuerdo con lo expuesto, modifique el procedimiento *Timer1_Timer* como se indica a continuación.

```
Sub Timer1_Timer ()
  If (Despertador.Text = Time$ And DespertadorSi) Then
```

```
    Beep: Beep: Beep
  End If
  Hora.Caption = Time$
End Sub
```

Desarrollemos ahora el código para las órdenes del menú *Despertador*, empezando por la orden más sencilla, *Cerrar*. Haga clic en este menú y después en la orden *Cerrar*. A continuación escriba:

```
Sub Cerrar_Click ()
  End
End Sub
```

El título de la orden *Despertador No* ya indica que el despertador no está activado, lo que implica que la variable *DespertadorSi* tenga un valor **False**. Cuando el usuario haga clic sobre esta orden el título de la misma deberá cambiar para indicar *Despertador Si* y la variable *DespertadorSi* tomará el valor **True**, y al contrario.

Cambiar en ejecución una orden de un menú

La orden para activar y desactivar el despertador que acabamos de comentar tiene que ser cambiada durante la ejecución del programa. Para hacer esto, no tiene mas que asignar el título deseado a la propiedad **Caption** del control correspondiente, en nuestro caso al control denominado *DesperiadorSiNo*.

Haga clic en el menú *Despertador* y después en la orden *Despertador No*. A continuación escriba:

```
Sub DespertadorSiNo_Click ()
  If (DespertadorSi) Then
    DespertadorSi = FALSE
    DespertadorSiNo.Caption = "Despertador No"
  Else
    DespertadorSi = TRUE
    DespertadorSiNo.Caption = "Despertador Si"
  End If
End Sub
```

Observe que cuando el usuario haga clic sobre la orden para activar y desactivar el despertador, el título de la misma cambiará para indicar *Despertador No* si la variable *DespertadorSi* es **True**, y la variable *DespertadorSi* tomará el valor **False** al contrario.

Esto también podría haberse hecho diseñando el menú con las órdenes posibles, *Despertador Si* y *Despertador No*, estando una de ellas, la no activa, no visible, lo que puede hacerse poniendo su propiedad **Visible** a valor **False**. Suponga que ha construido este menú con las siguientes propiedades (los valores para las propiedades que no se especifican son los dados por defecto):

Caption	*Despertador No*	*Despertador Si*
CtlName	DespertadorSiNo	DespertadorSiNo
Index	0	1
Visible	True (señalada)	False (sin señalar)

El código para estos controles puede ser el siguiente:

```
Sub DespertadorSiNo_Click (Index As Integer)
  DespertadorSiNo(0).Visible = TRUE
  DespertadorSiNo(1).Visible = TRUE
  DespertadorSiNo(Index).Visible = FALSE
  DespertadorSi = Index - 1
End Sub
```

Las dos órdenes forman un array de controles denominado *DespertadorSiNo*. Inicialmente está visible la orden *Despertador No*. Cuando el usuario haga clic sobre ella, se invoca el procedimiento *DespertadorSiNo_Click* que recibe un valor 0 en **Index**. La ejecución de este procedimiento es como sigue: se hacen visibles las dos órdenes; seguidamente se hace no visible la orden que indica el índice, en este caso *Despertador No*, quedando visible la orden *Despertador Si*; la variable *DespertadorSi* toma el valor **Index** – 1, en este caso –1 (**True**); y al contrario.

También, si lo prefiere, podríamos poner la orden *Despertador No* con una marca ✓ cuando esté activa, y sin marca cuando no esté activa, lo que puede hacerse poniendo su propiedad **Checked** a valor **True** o **False**. Inicialmente la pondremos a valor **True**, lo que implica que la variable *DespertadorSi* valga **False**. De acuerdo con lo dicho, el procedimiento podría ser el siguiente:

```
Sub DespertadorSiNo_Click ()
  If (DespertadorSi) Then
    DespertadorSi = FALSE
    DespertadorSiNo.Checked = TRUE
  Else
```

```
      Despertador1 = TRUE
      DespertadorNo.Checked = FALSE
   End If
End Sub
```

Añadir órdenes a un menú

En Visual Basic es posible añadir una orden a un menú utilizando la sentencia **Load** y también se puede eliminar una orden de un menú utilizando la sentencia **Unload**. Estas sentencias ya se expusieron al hablar de aplicaciones con múltiples formas en este mismo capítulo, lo que apunta a que **Load** y **Unload** se van a poder utilizar con muchos controles.

Para poder realizar estas operaciones sobre un menú, las órdenes del mismo tienen que pertenecer a un array de controles. La razón es que una vez añadida una orden no podríamos añadir un nuevo procedimiento para la misma, mientras que con un array el procedimiento es común a todas las órdenes.

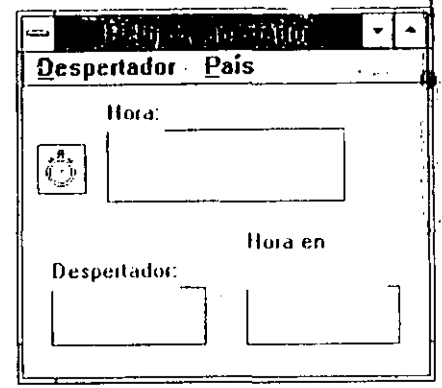
Quiere esto decir que durante el diseño hay crear un array de controles y esto exige que dicho array tenga al menos un elemento, al que uniremos el procedimiento común. Como inicialmente nosotros no sabemos qué órdenes van a añadirse al menú, la solución es que cuando diseñemos el menú hagamos que la primera orden, la de índice 0, sea invisible. De esta forma, las órdenes que se añadan después irán a continuación de esta invisible.

Para ver esto en la práctica, vamos añadir a nuestra aplicación, como ya se indicó en el enunciado, un segundo menú denominado *País* que permitirá al usuario añadir países al propio menú, para después haciendo clic sobre cualquiera de ellos, obtener en una tercera caja, formada por una etiqueta, la hora actual en ese país.

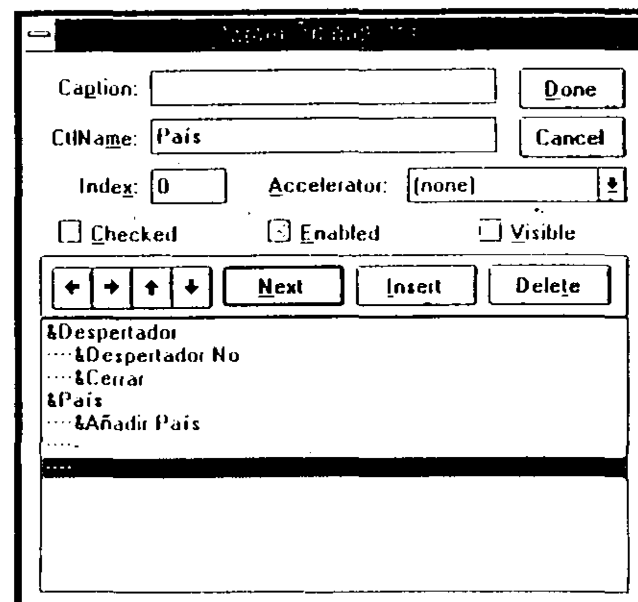
Lo primero que vamos a hacer es colocar esa tercera caja: haga un doble clic en la utilidad etiqueta del panel de utilidades y coloque la etiqueta en el lugar que se indica en la figura siguiente; ajuste su tamaño y asígnela las siguientes propiedades:

BorderStyle:	1	Estilo del borde
Caption	(nada)	Título
CtlName	OtraHora	Nombre del control
BackColor	&H0000FFFF&	Color de fondo
ForeColor	&H00000000&	Color del primer plano
FontSize	13.5	Tamaño del texto

Añada una etiqueta encima del borde de la caja anterior, no la asigne título, póngala el nombre *EtiquetaPaís* y ajuste el tamaño. Esta etiqueta la utilizaremos para visualizar el nombre del país del cual el usuario quiere saber su hora actual. Encima de esta etiqueta coloque otra con el título *Hora en*.



El paso siguiente es crear el menú *País* de forma que permita añadir nuevas órdenes. Abra la ventana de diseño de menús y escriba dicho menú como se ve en la figura siguiente.

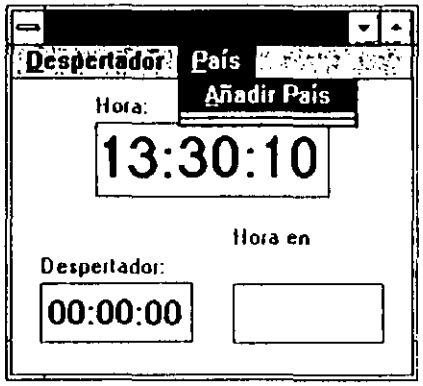


Inicialmente este menú se ha formado con las siguientes órdenes y propiedades:

Caption	*Añadir País*	- (guión)	(espacio en blanco)
CtlName	AñadirPaís	separador	País
Index	(ninguno)	(ninguno)	0
Visible	sí (señalada)	sí (señalada)	no (no señalada)

Observe en la figura anterior que *País(0)* es la primera orden de un array de controles, índice 0, y tiene un título, espacio en blanco, invisible (propiedad **Visible** sin señalar). En otras palabras, en nuestro array de controles podremos acceder desde el menú a los elementos *País(1)*, *País(2)*, y así sucesivamente, ya que el elemento *País(0)* permanece invisible.

Si ahora ejecuta el programa, puede ver que el menú aparece como en la figura siguiente.



Cuando el usuario haga clic sobre la orden *Añadir País* lo que se pretende es que pueda introducir dos nuevos datos: uno el nombre del país que será el título de la nueva orden y otro la diferencia horaria con respecto a nuestro país, para posteriormente poder calcular la hora actual en el país especificado.

Para tener acceso a estos datos, los guardaremos en un array de estructuras denominado *PaísHora*, donde cada elemento será del siguiente tipo:


```
Type TipoPaís
  País As String * 21
  Dif As Integer
End Type
Global PaísHora(20) As TipoPaís
```

Otro dato al que tenemos que seguir la pista es al número de elementos del menú, que nos va a proporcionar el valor del índice de la orden a añadir y que coincide con el índice del elemento del array donde almacenamos los datos. Para ello, declararemos la variable *NumPaíses* con un valor inicial 0, valor que ya tiene por defecto.

```
Global NumPaíses As Integer
```

Escriba todas estas declaraciones en el módulo global *reloj.bas*.

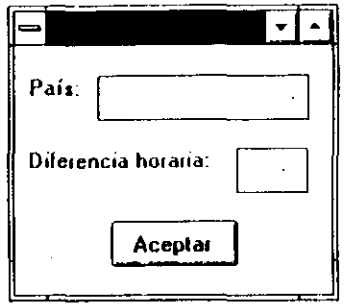
En conclusión, el conjunto de operaciones que tiene que suceder cuando el usuario haga clic sobre la orden *Añadir País* es:

1. Incrementar la variable *NumPaíses*.

   ```
   NumPaíses = NumPaíses + 1
   ```

2. Introducir los datos país y diferencia horaria.

   ```
   DatosEnt.Show 1
   ```



3. Cargar una nueva orden en el menú *País*.

   ```
   Load País(NumPaíses)
   ```

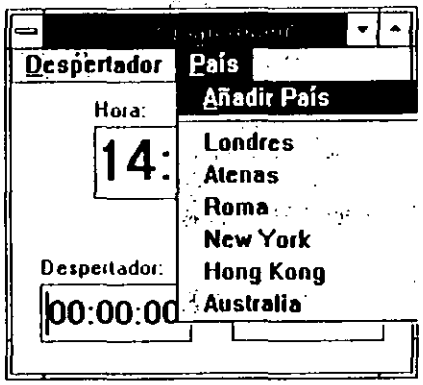
4. Asignar a la propiedad **Caption** de esta orden, el nombre del país y hacerla visible.

   ```
   País(NumPaíses).Caption = PaísHora(NumPaíses).País
   País(NumPaíses).Visible = True
   ```

En definitiva, el procedimiento *AñadirPaís_Click* queda como sigue:

```
Sub AñadirPaís_Click
  NumPaíses = NumPaíses + 1
  DatosEnt.Show 1
  Load País(NumPaíses)
  País(NumPaíses).Caption = PaísHora(NumPaíses).País
  País(NumPaíses).Visible = True
End Sub
```

Cuando ejecute esta orden y añada órdenes al menú *País*, el aspecto del menú será como el que se muestra a continuación.



Añadir una nueva forma

Disponer de más de una forma es particularmente útil cuando se necesita crear cajas de diálogo. En nuestro caso, según hemos visto en el apartado anterior, para añadir una nueva orden al menú *País* presentamos una segunda forma que representa una caja de diálogo la cual permite introducir los datos, país y diferencia horaria.

Continuando con nuestra aplicación, añada una nueva forma; para ello, ejecute la orden **New Form** del menú **File**. A continuación asigne a la forma el título **(Caption)** *Datos* y el nombre **(FormName)** *DatosEnt*.

El paso siguiente es añadir los controles a la forma. Fíjese en la forma *Datos* mostrada en el apartado anterior, añada los controles con las propiedades indicadas en la tabla siguiente, y ajuste el tamaño de los controles y de la forma.

Control	Caption	CtlName	Text
Etiqueta	País:	Label1	
Caja de texto		PaísEnt	(nada)
Etiqueta	Diferencia horaria:	Label2	
Caja de texto		DifHora	(nada)
Botón	Aceptar	Aceptar	

Una vez creada la forma, uniremos el código a cada uno de los controles. Cuando se ejecuta la orden *DatosEnt.Show 1* se muestra la forma titulada *Datos* con el cursor situado en la caja denominada *PaísEnt*. A continuación se espera que el usuario escriba el nombre del país que desea añadir y la diferencia horaria.

Una vez que el usuario ha tecleado los datos, si son correctos los aceptará pulsando el botón *Aceptar* y si no son correctos dirigirá de nuevo el cursor a la primera caja para escribirlos otra vez; en otras palabras, enfocará de nuevo la caja *PaísEnt*. Para enfocar un control puede utilizar el ratón o la tecla Tab.

Cuando se pulse el botón *Aceptar* queremos que sucedan tres cosas:

1. Almacenar el contenido de las cajas de texto, en el elemento correspondiente del array. Recuerde que el elemento del array se referencia por el mismo índice que la orden correspondiente del menú. Esto es, el elemento 1 del array contiene los datos para la orden *País(1)* del menú, el elemento 2 del array contiene los datos para la orden *País(2)* del menú, y así sucesivamente. Este índice viene dado por la variable global *NumPaíses*.
2. Que el cursor quede colocado en la caja *PaísEnt* para una próxima entrada.
3. Y que la forma quede oculta.

El procedimiento asociado se muestra a continuación.

```
Sub Aceptar_Click ()
  PaísHora(NumPaíses).País = PaísEnt.Text
  PaísHora(NumPaíses).Dif = Val(DifHora.Text)
  PaísEnt.SetFocus
  DatosEnt.Hide
End Sub
```

El método **SetFocus** permite enfocar el objeto especificado. Este objeto puede ser una forma o un control. Su sintaxis es:

objeto.**SetFocus**

Después de enfocar un objeto, cualquier entrada del usuario irá dirigida a ese objeto. Si en algún momento necesita saber qué objeto está enfocado, utilice las propiedades **ActivateControl** y **ActivateForm** del objeto **Screen**.

Screen.ActivateControl
Screen.ActivateForm

La propiedad **ActivateControl** devuelve el control que está enfocado y la propiedad **ActivateForm** devuelve la forma que está enfocada. Por ejemplo,

```
If Screen.ActiveControl.Caption = "Command1" Then
  Command2.SetFocus
Else
  Command1.SetFocus
End If
```

Este ejemplo dice que si el botón *Command1* está enfocado entonces pasar a enfocar el botón *Command2*, y si no está enfocado, enfocarlo.

Cuando un objeto se enfoca por cualquier acción, por ejemplo, haciendo clic sobre él, utilizando la tecla **Tab**, porque se ha cargado y visualizado una forma, o incluso porque se ha enfocado utilizando el método **SetFocus**, se da un suceso denominado **GotFocus** y cuando un objeto pierde el foco, se desenfoca, se da un suceso denominado **LostFocus**.

Como ejemplo, la sentencia *PaísEnt.SetFocus* del procedimiento *Aceptar_Click* presentado anteriormente, enfoca el objeto *PaísEnt*; esto implica que se dé el suceso *GotFocus*, que permite se ejecute el procedimiento *PaísEnt_GotFocus*, que se presenta a continuación. Como para cada entrada de datos el objetivo que se persigue es que el objeto enfocado sea la caja de texto *PaísEnt* y que ambas cajas de texto se presenten limpias de información, esto es, sin la información de la entrada anterior, escriba,

```
Sub PaísEnt_GotFocus ()
  PaísEnt.Text = ""
  DifHora.Text = ""
End Sub
```


Cada vez que se visualiza la forma, como el objeto enfocado es *PaísEnt*, se ejecuta este procedimiento que limpia las dos cajas de texto.

El botón *Aceptar* responde a un clic, a la tecla **Enter** o a la tecla **espacio**. Para hacer que respondara cualquier otra tecla, escriba el siguiente procedimiento:

```
Sub Aceptar_KeyDown (KeyCode As Integer, Shift As Integer)
  Aceptar_Click
End Sub
```

Procedimiento común para las órdenes añadidas

Cada una de las órdenes añadidas al menú *País*, permite saber la hora actual del país cuyo nombre coincide con el título de la orden. Cuando el usuario haga clic en alguna de esas órdenes, tiene que:

1. Mostrarse debajo de la etiqueta *Hora en*, otra etiqueta que especifique el nombre del país del cual queremos saber la hora.

   ```
   EtiquetaPaís.Caption = País(Index).Caption
   ```

2. Visualizarse en la caja de texto que hay debajo de la etiqueta anterior, denominada *OtraHora*, la hora actual correspondiente a ese país, la cual será la hora actual de nuestro país más la diferencia horaria.

El procedimiento común para estas órdenes que realiza lo anteriormente expuesto es el siguiente:

```
Sub País_Click (Index As Integer)
  Dim Horas As Integer
  EtiquetaPaís.Caption = País(Index).Caption
  Horas = Val(Hora.Caption) + País(Index).Dif H
  If Horas > 24 Then Horas = Horas - 24
  If Horas < 0 Then Horas = 24 - Horas
  OtraHora.Caption = Trim$(Str$(Horas) + Right$(Hora.Caption, 6)
  If Len(OtraHora.Caption) < 5 Then
    OtraHora.Caption = "0" + OtraHora.Caption
  End If
End Sub
```

El formato de *Hora.Caption* es *hh:mm:ss*. Por lo tanto la función **Val** devuelve el valor *hh*, al que se suma la diferencia horaria (positiva o negativa). Si el re-

sultado de esta operación es mayor que 24 la hora vendrá dada por la expresión *Horas − 24*, y si es negativo, por la expresión *24 + Horas*.

La expresión *Str$(Horas) + Right$(Hora.Caption, 6)* une la cadena "hh" con la cadena ":mm:ss".

La función **Ltrim$** elimina los espacios a la izquierda de una cadena de caracteres. La función **Str$** convierte un número a una cadena de caracteres y la función **Right$** toma los *n* caracteres más a la derecha de la cadena especificada.

Si la cadena resultante, debido a que la variable *Horas* tiene un dígito, es de la forma *h:mm:ss*, se añade un cero a la izquierda. La función **Len** da como resultado el número de caracteres de la cadena especificada.

Borrar órdenes de un menú

La sentencia **Unload** solamente puede utilizarse para borrar un objeto que haya sido creado con la sentencia **Load**; esto es, **Unload** no puede utilizarse para borrar un objeto creado en tiempo de diseño.

Por ejemplo, vamos a añadir a continuación de la orden *Añadir País* del menú *País* de nuestra aplicación, una orden titulada *Borrar País* y denominada *BorrarPaís*. El siguiente paso es escribir el procedimiento *BorrarPaís_Click* asociado con esta orden.

Cuando el usuario ejecute la orden *Borrar País*, Visual Basic invocará al procedimiento asociado *BorrarPaís_Click* que tiene que realizar lo siguiente:

1. Tomar el número (índice) de la orden del menú que se desee borrar.
2. Verificar que dicho número está dentro del rango.
3. Siendo *NumOrden* el número de la orden a borrar, reemplazar el título de la orden *NumOrden* con el título de la orden *NumOrden+1*, el título de la orden *NumOrden+1* con el título de la orden *NumOrden+2*, y así sucesivamente hasta el final del menú.
4. Borrar la última orden del menú.
5. Decrementar en una unidad la variable que cuenta el número total de órdenes.

El código correspondiente a lo expuesto es el siguiente:


```
Sub BorrarPaís_Click ()
  Dim NumOrden As Integer, N As Integer
  Dim Mensaje As String
  Mensaje = "Número del país a borrar:"
  NumOrden = Val(InputBox$(Mensaje
  If NumOrden > NumPaíses Or NumOrden < 1 Then
    MsgBox "Número fuera de rango"
    Exit Sub
  End If
  For N = NumOrden To NumPaíses - 1
    País(N).Caption = País(N + 1).Caption
    PaísHora(N).País = PaísHora(N + 1).País
    PaísHora(N).Diff = PaísHora(N + 1).Diff
  Next N
  Unload País(NumPaíses)
  PaísHora(NumPaíses).País = ""
  PaísHora(NumPaíses).Diff = 0
  NumPaíses = NumPaíses - 1
End Sub
```

La función **InputBox$** devuelve como resultado una cadena de caracteres que el usuario escribe como respuesta a una caja de diálogo que se visualiza cuando se ejecuta la función.

La función **MsgBox** visualiza en una caja de diálogo el mensaje especificado.

Tanto **InputBox$** como **MsgBox** serán vistas con más detalle en el capítulo siguiente.

CAPÍTULO 7

CAJAS DE DIÁLOGO

INTRODUCCIÓN

Visual Basic tiene una serie de controles para ser utilizados como mecanismos de entrada/salida (E/S). Un buen ejemplo de lo dicho son las cajas de texto que hasta ahora hemos venido utilizando. Otra serie de alternativas por las que podemos optar son: *señales* (cajas para señalar una opción), *opciones* (elegir una opción entre varias), *listas*, *combinados* y *barras de desplazamiento*.

E/S DE DATOS CON InputBox$ y MsgBox

Una ventana para entrada/salida de datos es útil cuando se necesita preguntar al usuario cualquier tipo de información, o cuando se necesita visualizar cualquier resultado o mensaje. Para crear una ventana de este tipo siga los siguientes pasos:

1. *Seleccione un suceso.* Lo primero es determinar qué suceso tiene que ocurrir para preguntar al usuario una determinada información, o para visualizar un resultado o un determinado mensaje.

   Por ejemplo, si desea que el usuario introduzca una palabra de paso (una clave) cuando intente ejecutar una determinada aplicación, piense en el suceso **Load**. Entonces, tiene que escribir el código necesario en el procedimiento *Form_Load* de la forma que se carga en el momento que se ejecuta dicha aplicación. En el supuesto de que el usuario introduzca una palabra de paso incorrecta, se le notificará con un mensaje.

2. *Escriba el código.* El código para una entrada, tiene que cargar y visualizar la ventana, y especificar con un mensaje el tipo de información o requiere

del usuario; y para una salida, tiene que cargar y visualizar la ventana, y presentar en la misma el resultado o el mensaje.

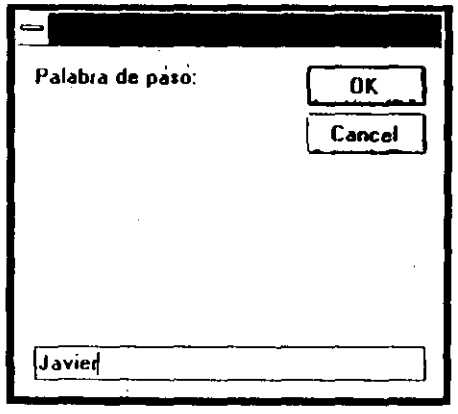
Visual Basic provee algunas ventanas predefinidas que nosotros podemos utilizar para este propósito: **InputBox$** y **MsgBox**.

La función **InputBox$** visualiza una caja de diálogo con un mensaje, semejante a la de la figura siguiente, que indica al usuario el tipo de información que debe introducir. Su sintaxis es:

**InputBox$**(*mensaje$*[, *título$*[, *omisión$*[, *posx%*, *posy%*]]])

La cadena de caracteres *mensaje$* contiene el mensaje que indica al usuario el tipo de información que debe introducir. Si el mensaje consiste de varias líneas, hay que introducir explícitamente al final de cada una de ellas, los caracteres retorno de carro (**Chr$(13)**) y avance de línea (**Chr$(10)**).

El resto de los parámetros son opcionales. La cadena de caracteres *título$* será visualizada en la barra de título de la caja de diálogo; *omisión$* es la entrada por defecto; y *posx* y *posy* son las coordenada *x* e *y* medidas desde la izquierda y desde la parte superior de la pantalla respectivamente y expresadas en **twips** (1/1440").

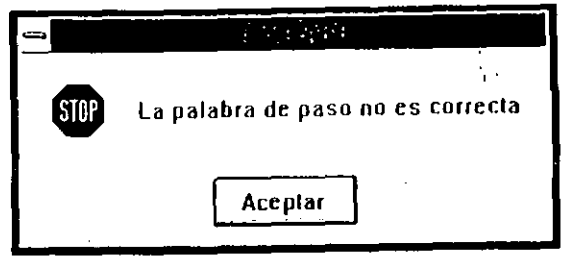


La sentencia o la función **MsgBox** visualizan un mensaje en una caja de diálogo, semejante a la de la figura siguiente. Su sintaxis es la siguiente:

MsgBox *mensaje$*[, *tipo%*[, *título$*]]
ValorRetornado% = **MsgBox**(*mensaje$*[, *tipo%*[, *título$*]])

La cadena de caracteres *mensaje$* contiene el mensaje que se desea visualizar.

El resto de los parámetros son opcionales. El entero *tipo* es una suma de valores que describe el número y el tipo de botones a visualizar, el estilo del icono, y la identificación del botón por defecto. Para ver estos valores así como los retornados por la función, vea la ayuda que Visual Basic da para **MsgBox**. La cadena de caracteres *título$* será visualizada en la barra de título de la caja de diálogo.



El código para el ejemplo expuesto y que da lugar a las cajas de diálogo anteriores es el siguiente:

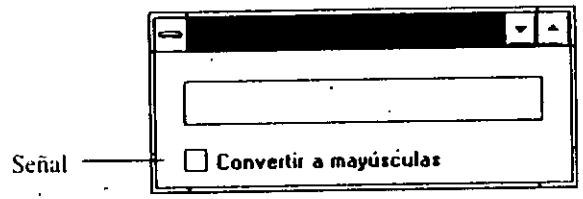
```
Sub Form_Load ()
  Dim Clave As String * 8
  Dim Cuenta As Integer
  Do Until UCase$(Clave) = "JAVIER  " Or Cuenta = 3
    Clave = InputBox$("Palabra de paso:")
    If UCase$(Clave) <> "JAVIER  " Then
      MsgBox "La palabra de paso no es correcta", 16
    End If
    Cuenta = Cuenta + 1
  Loop
End Sub
```

La variable *Clave* es una cadena fija de ocho caracteres que contendrá la palabra de paso que teclee el usuario (si no teclea ocho caracteres, Visual Basic la completa con blancos). Hasta que esa palabra de paso no coincida con la cadena de ocho caracteres "JAVIER " (hay dos espacios en blanco al final), y hasta un máximo de tres veces, el usuario será reiteradamente preguntado. Cada vez que no haya coincidencia se visualizará la caja de diálogo presentada anteriormente. El valor 16 de la sentencia **MsgBox** da lugar al icono STOP. Si hubiéramos escrito un valor 20 (16+4) se visualizarían además del icono, dos botones *Sí* y *No*, en lugar del botón *Aceptar*. En este último caso, si necesitamos saber qué botón se pulsó, utilizaremos la función en lugar de la sentencia, ya que la función devuelve como resultado el valor (6 o 7) correspondiente al botón pulsado.

SEÑALES - ☐

Una *señal* es un control que indica si una determinada opción está activada o desactivada. Cada opción es independiente de las demás ya que cada una de ellas tiene su propio nombre (propiedad **CtlName**). El número de opciones representadas de esta forma puede ser cualquiera, y de ellas el usuario puede seleccionar **todas las que desee** cada vez.

Si en tiempo de ejecución se hace clic sobre una opción simbolizada por un control de este tipo, ésta queda seleccionada (☒). Una opción ya seleccionada, puede pasar a no estarlo haciendo clic de nuevo sobre la opción (☐). Todo esto, suponiendo que el convenio adoptado es: una × opción seleccionada y nada opción no seleccionada.



Para saber si una determinada opción está seleccionada hay que verificar el valor de su propiedad **Value**. Este valor puede ser 0, la caja aparece vacía; 1, la caja aparece con una ×; y 2 la caja aparece en gris.

Cuando una de estas opciones está deshabilitada, aparece en gris. Esto se consigue poniendo su propiedad **Enabled** a valor 0 y es entonces cuando la propiedad **Value** vale 2.

Por ejemplo, diseñe una caja de diálogo como la de la figura anterior, de tal forma que cuando se seleccione la opción "Convertir a mayúsculas",

1. Todo el texto que se haya escrito en la caja de texto aparezca en mayúsculas.

2. Y todo el texto que se escriba a continuación, también aparezca en mayúsculas.

Suponiendo que la caja de texto se denomina *Text1*, que la opción "Convertir a mayúsculas" se denomina *ConverMayus*, y sabiendo que la opción se selecciona ha[illegible]o clic o eligiéndola con la tecla **Tab** y pulsando espacio, el primer punto se [illegible]e con el siguiente procedimiento:


```
Sub ConverMayus_Click ()
  If ConverMayus.Value = 1 Then
    Text1.Text = UCase$(Text1.Text)
  End If
  If ConverMayus.Value <> 2 Then
    Text1.SetFocus
  End If
End Sub
```

Cuando el usuario haga clic sobre la opción "Convertir a mayúsculas" y la propiedad **Value** valga 1, el texto de la caja *Text1* se convierte a mayúsculas utilizando la función **UcaseS**. Como la opción queda enfocada cada vez que se hace clic sobre ella, ejecutamos a continuación el método **SetFocus** para que sea el control *Text1* el que quede enfocado. Este método no puede ejecutarse si la opción está deshabilitada.

El segundo punto se resuelve recordando que cada vez que el usuario pulsa una tecla, uno de los sucesos que se da es **KeyPress** y que el parámetro **KeyAscii** de éste, contiene el valor ASCII del carácter pulsado. Entonces, lo que hay que hacer es convertir el valor ASCII al carácter correspondiente, convertir este carácter a mayúscula y convertir de nuevo éste último a valor ASCII.

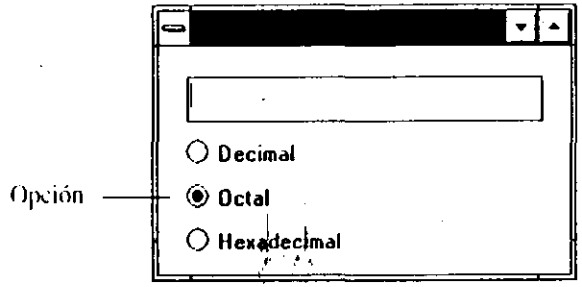
```
Sub Text1_KeyPress (KeyAscii As Integer)
  Dim car As String * 1
  If ConverMayus.Value = 1 Then
    car = UCase$(Chr$(KeyAscii))
    KeyAscii = Asc(car)
  End If
End Sub
```

La función Asc da como resultado el valor ASCII de carácter especificado.

OPCIONES - ○

Una *opción* es un control que indica si una determinada opción está activada o desactivada. Cada opción es independiente de las demás ya que cada una de ellas tiene su propio nombre (propiedad **CtlName**). El número de opciones representadas de esta forma puede ser cual[illegible]a y de ellas el usuario **sólo puede seleccionar una** cada vez.

Si en tiempo de ejecución se hace clic sobre una opción simbolizada por un control de este tipo, la opción queda seleccionada (•). La selección de una opción de este tipo, provoca que la opción actualmente seleccionada pase [illegible] estarlo.



Para saber si una determinada opción está seleccionada hay que verificar el valor de su propiedad **Value**. Este valor puede ser falso (0), el círculo aparece vacío, o verdadero (-1), el círculo aparece con un •. Cuando una opción se pone a valor verdadero, se da el suceso **Click**.

Cuando una de estas opciones está deshabilitada, aparece en gris. Esto se consigue poniendo su propiedad **Enabled** a valor 0.

Por ejemplo, diseñe una caja de diálogo como la de la figura anterior, de tal forma que cuando seleccione una de las opciones "Decimal", "Octal" o "Hexadecimal", el número tecleado en la caja de texto, aparezca en la base indicada. La tabla de propiedades puede ser la siguiente:

Control	Caption	CtlName	Value
Caja de texto		Text1	
Opción primera	Decimal	Decimal	True (-1)
Opción segunda	Octal	Octal	False (0)
Opción tercera	Hexadecimal	Hexadecimal	False (0)

Esta aplicación responde a los siguientes sucesos:

- Cada vez que el usuario modifique el contenido de la caja de texto se dará el suceso **Change**. En este instante almacenaremos el valor decimal, octal o hexadecimal (dependiendo de la opción seleccionada) en una variable entera denominada *NúmeroActual*.

```
Sub Text1_Change ()
  Texto = LTrim$(Text1.Text)
  If Decimal.Value And Not (Car = "-" Or Car = "+") Then
    NúmeroActual = Val(Texto + "&")
  ElseIf Octal.Value Then
    NúmeroActual = Val("&O" + Texto + "&")
  ElseIf Hexadecimal.Value Then
    NúmeroActual = Val("&H" + Texto + "&")
  End If
End Sub
```


La función **Val** convierte una cadena de caracteres a un número. Si la cadena comienza con los caracteres "&O" será interpretada como un valor en octal; si comienza con los caracteres "&H" será interpretada como un valor en hexadecimal; y si comienza por los caracteres "&D", por +, por − o simplemente por un dígito será interpretada como un valor en decimal. En este último caso, si la cadena tiene aún un sólo carácter, + o −, eludimos la conversión a decimal, de lo contrario obtendremos un error. El sufijo "&" especifica que ese valor es un entero.

- Cuando el usuario seleccione una de las opciones *Decimal*, *Octal* o *Hexadecimal*, se dará el suceso **Click** asociado con ese control que hará que el valor *NúmeroActual* se visualice en decimal, octal o hexadecimal respectivamente. Como la opción una vez seleccionada queda enfocada, ejecutamos a continuación el método **SetFocus** para que sea el control *Text1* el que quede enfocado. Este método no puede ejecutarse si la opción está deshabilitada.

```
Sub Decimal_Click ()
  Text1.Text = Format$(NúmeroActual)
  Text1.SetFocus
End Sub

Sub Octal_Click ()
  Text1.Text = Oct$(NúmeroActual)
  Text1.SetFocus
End Sub

Sub Hexadecimal_Click ()
  Text1.Text = Hex$(NúmeroActual)
  Text1.SetFocus
End Sub
```

La función **Oct$** devuelve una cadena de caracteres correspondiente a la representación octal del valor especificado y la función **Hex$** devuelve una cadena de caracteres correspondiente a la representación hexadecimal del valor especificado.

Con el fin de limitar la entrada a los caracteres que forman parte de cada base, escriba el procedimiento siguiente:

```
Sub Text1_KeyPress (KeyAscii As Integer)
  Car = UCase$(Chr$(KeyAscii))
  If Decimal.Value Then
    vr = InStr(Deci, Car)
  ElseIf Octal.Value Then
    vr = InStr(Octa, Car)
  ElseIf Hexadecimal.Value Then
    vr = InStr(Hexa, Car)
  End If
  If vr = 0 Then
    Beep
    KeyAscii = 0
  End If
End Sub
```

La función **InStr**(*XS, YS*) devuelve la posición del primer carácter de *YS* en *XS*. Si *YS* no se encuentra en *XS*, la función devuelve como resultado 0.

Las variables que aparecen en este procedimiento y en los anteriores están definidas en la sección **(general)** de la forma, y son inicializadas por el procedimiento *Form_Load* como se indica a continuación.

```
Dim NúmeroActual As Integer
Dim Deci As String * 12
Dim Octa As String * 10
Dim Hexa As String * 18
Dim Car As String * 1, vr As Integer, Texto As String

Sub Form_Load ()
  Deci = "--0123456789" + Chr$(8) + Chr$(13)
  Octa = "01234567" + Chr$(8) + Chr$(13)
  Hexa = "0123456789ABCDEF" + Chr$(8) + Chr$(13
End Sub
```

Las expresiones *Chr$(8)* y *Chr$(13)* se corresponden con los caracteres de retroceso (**BackSpace**) y retorno de carro respectivamente.

LISTAS Y COMBINADOS

Una *lista* es un control que pone a disposición del usuario un conjunto de elementos, de los cuales elegirá uno. Si el número de elementos rebasa el número de los que pueden ser visualizados simultáneamente en el espacio disponible en la *lista*, el usuario puede desplazar la lista de elementos hacia arriba o hacia abajo. Generalmente, una lista es apropiada cuando se quiere limitar la entrada a una serie de elementos determinados.

Un *combinado* es un control que combina las características de una caja de texto y de una lista. Esto permite al usuario elegir un elemento de varios, escribiéndolo directamente en la caja de texto o seleccionándolo de la lista. Generalmente, un control combinado es apropiado cuando hay una lista sugerida de elementos y además, el usuario puede sugerir otros que no están en la lista.

Para colocar un elemento en una lista o en un combinado, utilice el método **AddItem** y para eliminar un elemento de alguno de estos controles, utilice el método **RemoveItem**.

Para acceder a los elementos de una lista o de un combinado, puede utilizar alguna de las propiedades siguientes: **Text**, **List**, **ListIndex**, o **ListCount**. De todas ellas, la más sencilla de utilizar es **Text**. Esta propiedad proporciona la cadena de caracteres correspondiente al elemento elegido.

Los sucesos más importantes para una lista son **Click** (clic) y **DblClick** (doble clic) y para un combinado, **Click** y **Change**.

Utilización de listas

Supongamos una pequeña base de datos para llevar la cuenta de los libros de nuestra biblioteca particular, de los cuales, hemos prestado algunos a otras personas. Para seguir la pista a estos libros vamos a almacenar los siguientes datos:

Título de libro
Autor
Editorial
Datos sobre el préstamo

Cada uno de estos datos elementales se denomina *campo* y el conjunto de todos los campos referentes a un mismo libro recibe el nombre de *registro*.

Nuestra aplicación va a consistir de una ventana principal que permita introducir o visualizar los datos de un registro y de un menú que permita, entre otras cosas, buscar un determinado registro. Cuando el usuario seleccione en este menú la orden "Buscar Registro..." aparecerá una caja de diálogo con una lista clasificada de los títulos de los libros prestados. Cuando el usuario haga un doble clic sobre uno de los títulos, los datos correspondientes a ese libro se visualizarán en la ventana principal.

Generalmente una base de datos está clasificada por alguno de sus campos, en nuestro caso lo va a estar por el campo "Título". Para realizar esta operación nos va a ser de gran ayuda la propiedad **Sorted**: cuando esta propiedad tenga valor **True** (-1), Visual Basic visualizará la lista clasificada.

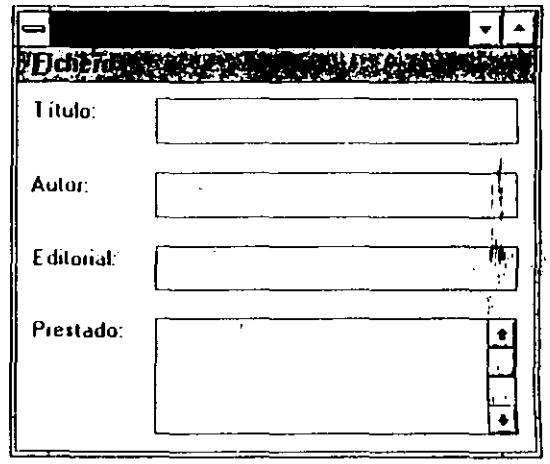
Para diseñar esta aplicación, comience un nuevo proyecto y asigne a la forma el nombre *Libros Prestados*. A continuación añada cuatro cajas de texto, cada una de las cuales se corresponderá con un campo del registro, y cuatro etiquetas que las identifiquen.

Control	**Caption**	**CtlName**
Caja de texto 1		Título
Caja de texto 2		Autor
Caja de texto 3		Editorial
Caja de texto 4		Prestado
Etiqueta 1	Título:	
Etiqueta 2	Autor:	
Etiqueta 3	Editorial:	
Etiqueta 4	Prestado:	

Haga que la caja de texto *Prestado* sea una caja multilínea y con una barra de desplazamiento vertical. Esto le permitirá introducir varias líneas de información. Para ello, ponga la propiedad **Multiline** a valor **True** y la propiedad **ScrollBars** a valor 2.

Ahora añada un menú denominado *Fichero* con la órdenes, *Añadir Registro*, *Buscar Registro...* y *Salir*. Asigne a estas órdenes los nombres *AñadirReg*, *BuscarReg* y *Salir* respectivamente.

Ejecute la orden **Save Project** para guardar el trabajo realizado hasta ahora y asigne a la forma el nombre *libros.frm* y a la aplicación el nombre *libros.mak*.



En primer lugar vamos a escribir el código correspondiente a la orden *Añadir Registro* del menú *Fichero*. Cuando el usuario haga clic en esta orden deseará que el contenido actual de las cajas de texto sea almacenado en la base de datos. Nuestra base de datos va a estar formada por un array de registros denominado *Libros*, que lo definiremos, como se indica a continuación, en el módulo global *libros.bas*. En el próximo capítulo veremos como almacenar estos datos en un fichero.

```
Type Registro
  Título As String * 30
  Autor As String * 30
  Editorial As String * 12
  Prestado As String * 240
End Type
Global Libros(1 To 40) As Registro
Global TotalRegs As Integer
```

Visualice el módulo **global.bas**, escriba las declaraciones y definiciones anteriores y guarde de nuevo el módulo, ahora con el nombre *libros.bas*, para lo cual tiene que ejecutar la orden **Save File As...** del menú **File**.

Un array definido de esta forma puede ser accedido desde cualquier procedimiento de la aplicación. Observe que el array puede tener un máximo de 40 registros. Para llevar la cuenta de los registros que hay almacenados, se ha definido la variable *TotalRegs*. El valor inicial de esta variable es 0.

Para acceder al primer registro del array, *Libros(1)*, primero incremente la variable *TotalRegs*; después almacene la información de cada una de las cajas de texto en los campos correspondientes del registro. A continuación, hay que añadir

el *Título* a la lista que aparecerá en la caja de diálogo que se visualiza cuando se ejecuta la orden *Buscar Registro*. Para los siguientes registros del array se procede de la misma forma. El procedimiento completo se muestra a continuación.

```
Sub AñadirReg_Click ()
  TotalRegs = TotalRegs + 1
  Libros(TotalRegs).Título = Título.Text
  Libros(TotalRegs).Autor = Autor.Text
  Libros(TotalRegs).Editorial = Editorial.Text
  Libros(TotalRegs).Prestado = Prestado.Text
  Form2.ListaLibros.AddItem Título.Text
End Sub
```

El método **AddItem** permite añadir un elemento a una lista (lista o combinado). Su sintaxis es la siguiente:

nombre-lista.**AddItem** *elemento*[, *índice*]

donde *nombre-lista* es el nombre del control (lista o combinado) y *elemento* es una cadena de caracteres correspondiente al elemento a añadir. El argumento *índice*, si se especifica, indica la posición donde se insertará el nuevo elemento. Un valor 0 indica la primera posición. Si *índice* no se especifica, el elemento se añade al final de la lista.

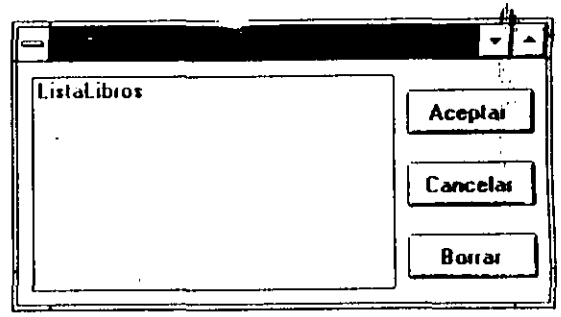
La lista se visualizará clasificada si la propiedad **Sorted** del control se pone a valor **True** (-1).

Siguiendo con nuestra aplicación, el paso siguiente es añadir una nueva forma que contenga la caja de diálogo que se tiene que visualizar cuando se ejecute la orden *Buscar Registro*.

Cree una nueva forma, *Form2*, y asígnela el nombre *Buscar Registro*. Después, elimine los botones que permiten maximizar y minimizar la ventana, poniendo a valor **False** (0) las propiedades **MaxButton** y **MinButton** respectivamente y fije el borde, asignando un valor 3 a la propiedad **BorderStyle**, para que no se pueda modificar el tamaño de la forma.

A continuación añada los controles que se especifican en la tabla siguiente. El resultado se ve en la figura que se muestra a continuación.

Control	Caption	CtlName	Sorted
Lista		ListaLibros	True
Botón 1	Aceptar	Aceptar	
Botón 2	Cancelar	Cancelar	
Botón 3	Borrar	Borrar	



Escribamos ahora el código para la orden *Buscar Registro*. Cuando el usuario haga clic en esta orden se tiene que visualizar la caja de diálogo presentada en la figura anterior, lo que se consigue ejecutando el método **Show** para esta forma.

```
Sub BuscarReg_Click ()
  Form2.Show
  Form2.Cancelar.SetFocus
End Sub
```

Cuando se visualice la forma *Buscar Registro*, aparecerá sobre ella la lista de libros clasificada y el botón *Cancelar* enfocado. Ahora, el usuario seleccionará el libro que busca, del que quiere conocer el resto de los datos. Una vez seleccionado, hará clic en el botón *Aceptar* para que los datos relativos a dicho libro se visualicen en la ventana principal. También, en lugar de hacer clic sobre el botón *Aceptar*, el usuario podría optar por hacer un doble clic sobre el elemento seleccionado, obteniendo el mismo resultado.

Quiere esto decir, que tanto el procedimiento ligado al botón *Aceptar* como el procedimiento ligado a la lista para un doble clic, tienen que invocar a un mismo procedimiento, *VisualizarRegistro*, que se encargue de buscar el registro en la base de datos y de visualizarlo en la ventana principal.

```
Sub Aceptar_Click ()
  Call VisualizarRegistro
End Sub
```

```
Sub ListaLibros_DblClick ()
  Call VisualizarRegistro
End Sub
```

Para que un procedimiento (**Sub o Function**) pueda ser invocado desde cualquier parte de la aplicación, hay que colocarlo en un módulo, proceso que realizamos más adelante.

Una vez presentada la forma *Visualizar Registro* el usuario se puede encontrar con que el libro que busca no está en la lista y debido a ello, simplemente, quiere abandonar esta caja de diálogo, para lo que pulsará el botón *Cancelar*. Quiere esto decir, que el procedimiento ligado al botón *Cancelar* tiene que ocultar la forma *Form2*.

```
Sub Cancelar_Click ()
  Form2.Hide
End Sub
```

La función de la orden *Salir* es finalizar la aplicación, cosa que se consigue ejecutando la sentencia **End**.

```
Sub Salir_Click ()
  End
End Sub
```

Crear un módulo

Para crear un módulo, seleccione la orden **New Module** del menú **File** y a continuación siga uno de los dos caminos a) o b) que se indican a continuación:

a) En la ventana de código que aparece escriba el procedimiento, incluyendo las líneas **Sub** y **End Sub** del mismo.

b) Ejecute la orden **New Procedure** del menú **Code**. En la ventana de diálogo que se presenta, elija la opción **Sub** y escriba en la caja de texto el nombre del procedimiento *VisualizarRegistro*. Finalmente, pulse el botón **OK** y escriba el cuerpo del procedimiento entre **Sub** y **End Sub**.

Para modificar un módulo existente, selecciónelo en la ventana de la aplicación, pulse el botón **View Code**, y seleccione el procedimiento en la caja de procedimientos **Proc:** de la ventana de código presentada.

Siguiendo con la aplicación y teniendo en cuenta lo expuesto anteriormente, el procedimiento *VisualizarRegistro* es el siguiente:

```
Sub VisualizarRegistro ()
  Dim I As Integer
  For I = 1 To TotalFega
    If (RTrim$(Libros(I).Título) =
        RTrim$(Form2.ListaLibros.Text)) Then
      Exit For
    End If
  Next I
  Form1.Título.Text = Libros(I).Título
  Form1.Autor.Text = Libros(I).Autor
  Form1.Editorial.Text = Libros(I).Editorial
  Form1.Prestado.Text = Libros(I).Prestado
  Form2.Hide
End Sub
```

Observe que el procedimiento anterior realiza una búsqueda secuencial a partir del registro 1, para hallar la posición *I* en la que se encuentra el registro correspondiente al libro buscado.

Tenga en cuenta que al ser las cadenas de caracteres de los registros del array de longitud fija, Visual Basic las completa con blancos. Dos cadenas de caracteres con la única diferencia de que una de ellas tenga al final más espacios en blanco que la otra, son desiguales. Por ello, para evitar resultados inesperados, utilice la función **RTrim$** que elimina los espacios en blanco a la derecha de una cadena de caracteres.

Eliminar un elemento de una lista

Para finalizar nuestra aplicación queda por escribir el procedimiento asociado con la orden *Borrar* de la caja de diálogo *Buscar Registro*.

Para eliminar un elemento de una lista o de un combinado se utiliza el método **RemoveItem**. La sintaxis es la siguiente:

nombre-lista.**RemoveItem** *índice*

donde *nombre-lista* es el nombre del control (lista o combinado) y el argumento *índice* indica la posición del elemento que se desea eliminar. Un valor 0 indica la primera posición.

La propiedad **ListIndex** da la posición, respecto de 0, del elemento actualmente seleccionado. También permite fijar dicha posición. Su sintaxis es:

[*forma.*]*control*.**ListIndex**[= *posición%*]

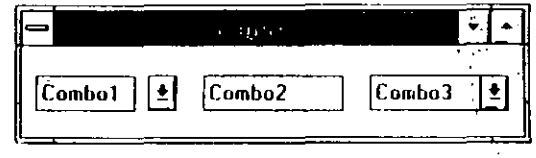
Cuando el usuario pulse el botón *Borrar*, el elemento seleccionado tiene que borrarse de la lista *ListaLibros* y del array *Libros*. Tenga en cuenta que la lista está clasificada y el array que forma la base de datos no lo está; quiere esto decir que un determinado elemento generalmente ocupará posiciones diferentes en la lista y en el array. Para borrar el registro del array correspondiente al elemento seleccionado de la lista, primero realizamos una búsqueda secuencial para localizarlo y después lo borramos escribiendo el siguiente registro sobre él, lo que implica retroceder una posición todos los registros que estén a continuación del que queremos borrar. Y para borrar el elemento seleccionado de la lista, calculamos su posición por medio de la propiedad **ListIndex** y lo eliminamos utilizando el método **RemoveItem**.

El siguiente procedimiento realiza estas dos funciones.

```
Sub Borrar_Click ()
  Dim I As Integer, R As Integer
  'Borrar el registro del array
  'Se busca
  For I = 1 To TotalRegs
    If RTrim$(Libros(I).Título) =
       RTrim$(Form2.ListaLibros.Text) Then
      Exit For
    End If
  Next I
  If I > TotalRegs Then Exit Sub 'No encontrado
  'Se borra
  For R = I To TotalRegs - 1
    Libros(R) = Libros(R + 1)
  Next R
  Libros(TotalRegs).Título = ""
  Libros(TotalRegs).Autor = ""
  Libros(TotalRegs).Editorial = ""
  Libros(TotalRegs).Prestado = ""
  TotalRegs = TotalRegs - 1

  'Borrar el título de la lista
  'Posición en la lista
  I = Form1.ListaLibros.ListIndex
  'Se borra
  Form1.ListaLibros.RemoveItem I
End Sub
```


Utilización de combinados

La diferencia entre una *lista* y un *combinado* es que el combinado es una lista con una caja de texto en la que el usuario puede escribir su propio texto, correspondiente a un posible elemento, sin necesidad de seleccionar un elemento de la lista. Hay tres estilos diferentes de *combinados* los cuales se muestran en la figura siguiente como *Combo1*, *Combo2* y *Combo3*.



El control *Combo1* representa un combinado estándar y se obtiene poniendo la propiedad **Style**, asociada con dicho control, a valor 0. Cuando el usuario haga clic en la flecha, se visualizará la lista de elementos y podrá optar por elegir un elemento de la lista o por escribir directamente el elemento deseado en la caja de texto.

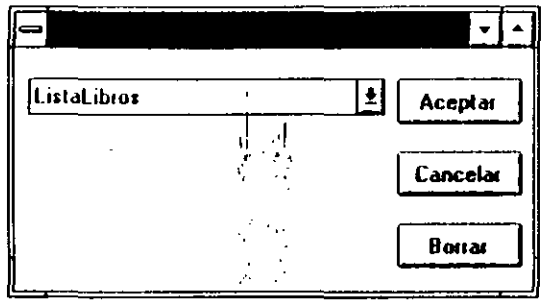
El control *Combo2* representa un combinado en el cual la lista de elementos siempre está visualizada y se obtiene poniendo la propiedad **Style**, asociada con dicho control, a valor 1. El usuario podrá optar por elegir un elemento de la lista o por escribir directamente el elemento deseado en la caja de texto.

El control *Combo3* representa un combinado que se diferencia del estándar en que el usuario sólo tiene la posibilidad de elegir un elemento de la lista, esto es, no se le permite escribir en la caja de texto. Se obtiene poniendo la propiedad **Style**, asociada con dicho control, a valor 2. En otras palabras, *Combo3* es una lista enrollable.

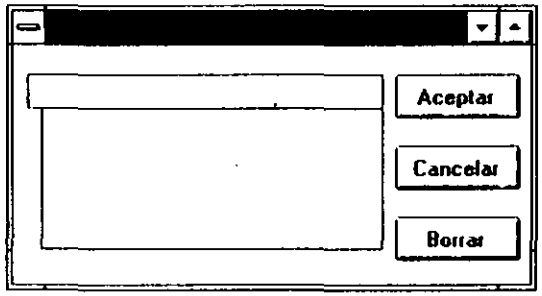
Para ver como trabaja este control vamos a modificar nuestra aplicación *Libros*, introduciendo en la caja de diálogo *Buscar Libro* un combinado en lugar de una lista.

La modificación mas sencilla es sustituir la lista por un combinado estilo 2. Para realizar esta operación seleccione la lista y bórrela, esto es, haga clic en el control *ListaLibros* y pulse la tecla **Supr** (**Del**). A continuación elija del panel de utilidades el control *combinado* y añádalo a la forma como se indica en la figura siguiente. Ponga la propiedad **Style** a valor 2, la propiedad **CtlName** a valor *ListaLibros* y la propiedad **Sorted** a valor **True**.

Ejecute ahora la aplicación y vea los resultados que se obtienen. Observe que el procedimiento *ListaLibros_DblClick* ahora no le sirve.



Modifique ahora la propiedad **Style** del combinado *ListaLibros* a valor 1 y ajuste el tamaño del control como se indica en la figura siguiente. Ahora tiene un combinado estilo 1. Ejecute ahora la aplicación y vea los resultados que se obtienen. Observe que el procedimiento *ListaLibros_DblClick* ahora sí le sirve. Ahora también puede escribir directamente en la caja de texto el elemento con el que desea operar, aunque no esté en la lista.



Por último, modifique la propiedad **Style** del combinado *ListaLibros* a valor 0. Ahora tiene un combinado estilo 0. Ejecute la aplicación y vea los resultados que se obtienen. Observe que el procedimiento *ListaLibros_DblClick* no le sirve. En este caso también puede escribir directamente en la caja de texto el elemento con el que desea operar, aunque no esté en la lista.

Para acceder a los elementos de una lista o de un combinado, puede utilizar alguna de las propiedades siguientes: **Text**, **List**, **ListIndex**, o **ListCount**. Las propiedades **Text** y **ListIndex** ya las hemos visto con las listas.

La propiedad **List** se corresponde con un array de cadenas de caracteres que contiene los elementos de la lista. Para acceder a uno de estos elementos hágalo como si de otro array se tratara. Así es,

[forma.]control.**List**(*índice%*[)] = *elementoS*]

El primer elemento tiene como *índice* 0 y el último **ListCount** – 1.

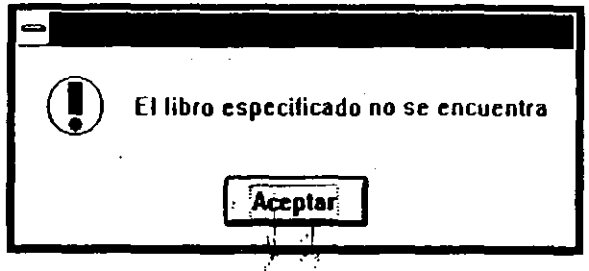
La propiedad **ListCount** da como resultado el número de elementos que hay en el control (lista o combinado).

Modifiquemos ahora nuestra aplicación para que cuando el usuario escriba en la caja de texto del combinado un elemento que no esté en la lista, se visualice una forma con un mensaje que lo indique. Como vimos al principio de este capítulo, esto se puede hacer fácilmente utilizando la sentencia **MsgBox**.

Para conseguir lo especificado, modifique el módulo *VisualizarRegistro* como se especifica a continuación.

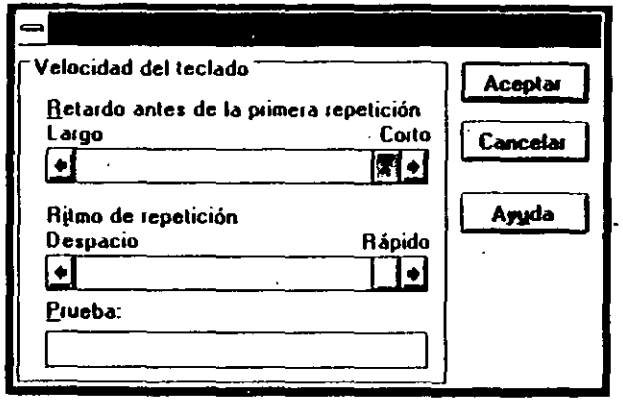
```
Sub VisualizarRegistro ()
  Dim I As Integer, Encontrado As Integer
  Encontrado = 0 'elemento no encontrado
  For I = 1 To TotalRegs
    If (RTrim$(Libros(I).Título) =
        RTrim$(Form2.ListaLibros.Text)) Then
      Encontrado = -1 'elemento encontrado
      Exit For
    End If
  Next I
  If Not Encontrado Then
    Form1.Título.Text = Libros(I).Título
    Form1.Autor.Text = Libros(I).Autor
    Form1.Editorial.Text = Libros(I).Editorial
    Form1.Prestado.Text = Libros(I).Prestado
    Form2.Hide
  Else
    MsgBox "El libro especificado no se encuentra", 48, "Libros"
  End If
End Sub
```

Observe que la información de si un determinado título se encuentra o no en la base de datos, nos la da la variable *Encontrado*. Cuando el título especificado en la caja de texto del combinado de estilo 0 o estilo 1 no se encuentre en la lista, la variable *Encontrado* valdrá 0 y se visualizará la forma que se presenta en la figura siguiente.



BARRAS DE DESPLAZAMIENTO

Las *barras de desplazamiento* son a menudo utilizadas en cajas de texto y ventanas para desplazar la información hacia abajo o hacia arriba de la ventana, o hacia la izquierda o hacia la derecha de la ventana. Pero también pueden utilizarse como controles de entrada como muestra la figura siguiente.



Una barra de desplazamiento representa un valor entero. Cada barra de desplazamiento tiene un botón que se desplaza a lo largo de la misma. La posición inicial se corresponde con el valor mínimo, la posición final se corresponde con el valor máximo y cualquier otra posición es un valor entre estos dos.

El valor mínimo se especifica mediante la propiedad **Min** y el valor máximo mediante la propiedad **Max**. Cualquier valor comprendido entre el mínimo y el máximo especificados, depende de la posición del botón que se desplaza a lo largo de la barra y viene dado por la propiedad **Value**.

Cuando el usuario hace clic encima o debajo del botón, este se desplaza una cantidad fija, negativa o positiva con respecto a la posición actual, que es especificada por la propiedad **LargeChange**. Cuando el usuario hace clic en la flecha que está al principio o al final de la barra, el botón se desplaza una cantidad fija, negativa o positiva con respecto a la posición actual, que es especificada por la propiedad **SmallChange**.

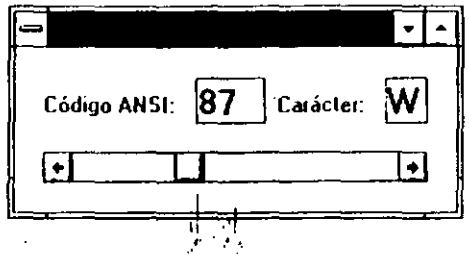
La unidad de medida utilizada por defecto por Visual Basic es el **twip** que equivale a 1/1440 pulgadas. El tamaño de una fuente se especifica en puntos, un punto se define como 1/72 pulgadas y un **twip** es 1/20 puntos. Un **twip** es una unidad de medida independiente de la pantalla lo que garantiza que la situación y la proporción de los objetos de una aplicación en la pantalla, sea la misma en todos los sistemas.

Cada vez que el usuario desplaza el botón a lo largo de la barra se da el suceso **Change**.

Como ejercicio vamos a construir una aplicación que permita visualizar el carácter correspondiente a un código ANSI en el rango de 0 a 255. Como se muestra en la figura siguiente, una barra de desplazamiento proporciona un valor de 0 a 255 que se visualiza en una etiqueta, y en otra etiqueta se visualiza el carácter correspondiente.

Inicie una nueva aplicación y asigne a la forma por defecto el título *ANSI*. A continuación ponga sobre la forma los controles con las propiedades que especifican en la tabla siguiente, ajuste los tamaños y guarde la aplicación con el nombre *barras.mak*.

	Etiqueta 1	Etiqueta 2	Etiqueta 3	Etiqueta 4	Barra de Desplaz.
Caption	Código ANSI:	0	Carácter:	(nada)	
CtlName	Label1	Código	Label2	Carácter	DespH
FontSize	8.5	13.5	8.5	13.5	
BorderStyle	0	1	0	1	
Min					0
Max					255
SmallChange					1
LargeChange					



Cuando el usuario actúe sobre la barra de desplazamiento *DespH*, se dará el suceso **Change** para este control, lo que hará que se ejecute el procedimiento asociado *DespH_Change* el cual tiene que visualizar el código ANSI en la etiqueta *Código* y el carácter correspondiente en la etiqueta *Carácter*.

```
Sub DesH_Change ()
  Código.Caption = Format$(DesH.Value)
  Carácter.Caption = Chr$(DesH.Value)
End Sub
```

MARCOS

Un marco tiene propiedades propias (por ejemplo, título, color) característica que aprovechamos cuando queremos realzar el aspecto de una forma. Normalmente, un marco se utiliza para agrupar objetos relacionados entre sí.

Para crear un marco, proceda como se indica a continuación:

1. Diríjase al panel de utilidades y haga un doble clic sobre la utilidad marco, o bien seleccione esta utilidad utilizando la tecla **Tab** y pulse **Enter**.
2. Ajuste el tamaño del marco y muévalo a la posición deseada. Puede hacerlo con el ratón o modificando las propiedades correspondientes.
3. Si lo desea, ponga un título al marco. Seleccione la propiedad **Caption** y escriba el título que desee.

Una vez creado el marco, añada los controles que van a estar agrupados bajo el mismo. Para añadir un control, diríjase al panel de utilidades, haga un clic sobre él, desplace el ratón (+) al interior del marco, y con el botón izquierdo del ratón pulsado arrastre hasta dibujarlo.

UN SHELL DE WINDOWS

La función **Shell** de Visual Basic permite ejecutar cualquier fichero ejecutable. Su sintaxis es la siguiente:

id = **Shell**(*orden$*[, *estilo%*])

La cadena de caracteres *orden$* es el nombre del programa a ejecutar incluyendo los argumentos en línea de órdenes. Este nombre corresponderá a un fichero existente en el disco con extensión .COM, .EXE, .BAT, o .PIF. El parámetro *estilo* determina el estilo de la ventana sobre la cual se va a ejecutar el programa. Los posibles valores para este parámetro son,

1	Ventana normal, enfocada
2	Ventana minimizada, enfocada
3	Ventana maximizada, sin enfocar
4	Ventana normal, sin enfocar
7	Ventana minimizada, sin enfocar

Si el programa se ejecuta satisfactoriamente, la función **Shell** retorna un valor único *id* que identifica al programa arrancado. En otro caso, Visual Basic genera un mensaje de error.

COLORES

Durante el diseño de una aplicación se puede añadir fácilmente color a las formas, a los controles y a los gráficos utilizando la paleta de colores de Visual Basic. La paleta de colores ofrece 48 colores fijos y 16 colores más que el usuario puede definir y añadir a la paleta.

Para poner color a un objeto durante el diseño, los pasos a seguir son los siguientes:

1. *Abrir la paleta de colores*. Para ello ejecute la orden **Color Palette** del menú **Window**.
2. *Seleccionar el objeto*. Puede seleccionar cualquier forma, control o gráfico haciendo clic sobre él.
3. *Elegir el color* de fondo y del primer plano (vea la paleta de colores en el Capítulo 1).

También durante la ejecución de una aplicación se puede modificar el color de cualquier objeto.

Por ejemplo, suponga una caja de texto denominada *Resultado* que se utiliza para visualizar resultados numéricos positivos o negativos. El siguiente procedimiento, ligado a esta caja y que se ejecuta cada vez que su contenido cambia, utiliza dos colores para diferenciar un valor positivo de otro negativo.

```
Sub Resultado_Change ()
  If Val(Resultado.Text) >= 0 Then
    Resultado.ForeColor = RGB(0, 0, 255) 'color azul
  Else
    Resultado.ForeColor = RGB(255, 0, 0) 'color rojo
  End If
End Sub
```

En este ejemplo, la sentencia **If** analiza el valor de la caja de texto *Resultado*: si es positivo asigna al primer plano (al texto) el color azul y si es negativo asigna al primer plano el color rojo.

Función RGB

La función **RGB** permite especificar un color cualquiera. Su sintaxis es la siguiente:

RGB(*rojo%*, *verde%*, *azul%*)

La función **RGB** tiene tres parámetros enteros en el rango de 0 a 255 que especifican el nivel de color rojo, verde y azul respectivamente. Esta función retorna un entero de cuatro bytes (entero largo) que representa el color RGB especificado. Este valor expresado de una forma genérica en hexadecimal es de la forma *&H00AAVVRR*. El byte más significativo es cero, y los tres bytes siguientes especifican el nivel de color *Azul*, *Verde* y *Rojo* respectivamente.

La siguiente tabla muestra los valores para algunos colores estándar.

Color	valor RGB	nivel de azul	nivel de verde	nivel de rojo
Negro	&H00	0	0	0
Azul	&HFF0000	255	0	0
Verde	&HFF00	0	255	0
Cyan	&HFFFF00	255	255	0
Rojo	&HFF	0	0	255
Magenta	&HFF00FF	255	0	255
Amarillo	&HFFFF	0	255	255
Blanco	&HFFFFFF	255	255	255

Visual Basic dispone también de la función **QBColor** y del método **Point**, que están directamente relacionados con la función **RGB**.

La función **QBColor** tiene un único argumento entero en el rango de 0 a 15 que se corresponde con un color estándar, y devuelve como resultado un entero largo correspondiente al color RGB.

```
Form1.BackColor = QBColor(1) 'color azul
```

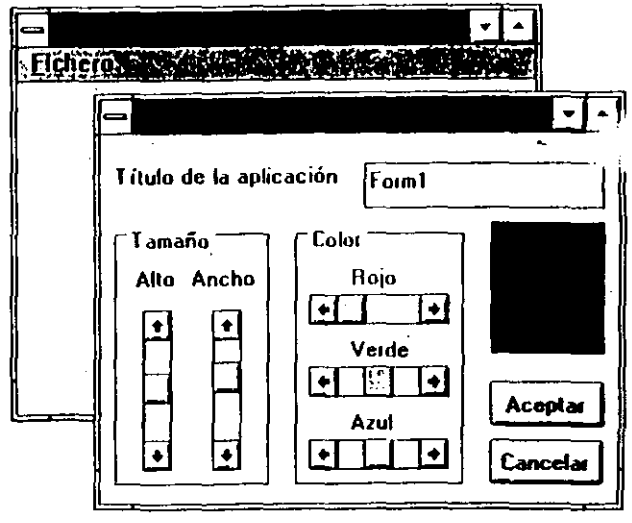
El método **Point** tiene dos argumentos de tipo real de precisión simple que especifican las coordenadas de un punto de una forma o de una imagen (control imagen) y que devuelve como resultado un entero largo correspondiente al color RGB que tiene el punto especificado. Las coordenadas están referidas a la forma.

```
Color = Form1.Point(0, 0)
```

APLICACIONES A MEDIDA

La siguiente aplicación presenta sobre la pantalla una simple ventana con un menú *Fichero*. Dicho menú está formado por tres órdenes: *Panel de Control*, *Ejecutar* y *Salir*.

Un panel de control sirve para definir ciertas características de una aplicación. En nuestro caso, nos va a permitir definir el tamaño, el color de fondo y el título de la ventana principal que por defecto es *Form1*. La figura siguiente muestra la ventana principal y el panel de control que queremos diseñar.



Para empezar cree una nueva aplicación, *panel.mak*, y añada a la forma *Form1* el menú *Fichero* con las órdenes que se indican en la siguiente tabla:

Caption	CtlName
Panel de Control...	FicheroPanel
Ejecutar...	FicheroEjecutar
Salir	FicheroSalir

Cuando el usuario haga clic en la orden *Salir* la aplicación tiene que finalizar. El procedimiento siguiente realiza esta operación.

```
Sub FicheroSalir_Click ()
  End
End Sub
```

El siguiente paso es diseñar el *Panel de Control*. Cree una nueva forma y asígnela el título *Panel de Control*, y el nombre (**FormName**) *PanelDeControl*.

Esta forma va a contener los siguientes controles: una caja de texto que permitirá al usuario introducir un nuevo nombre para la ventana principal. Dos barras de desplazamiento vertical que permitirán al usuario modificar el tamaño de la ventana principal. Recuerde que las medidas para una ventana hay que expresarlas en **twips**. Tres barras de desplazamiento horizontal que permitirán al usuario variar el color de fondo de la ventana principal. Un marco, *Muestra*, para ver una muestra del color seleccionado. Para resaltar las características del panel de control agruparemos las barras de desplazamiento de acuerdo con su función, para lo que utilizaremos dos marcos: *Tamaño* y *Color*. Un botón, *Aceptar*, que permitirá al usuario aceptar las características nombre, tamaño y color especificadas y otro botón, *Cancelar*, para poder retornar a la ventana principal sin realizar ninguna modificación.

Recuerde que para agrupar dentro de un marco un determinado número de controles, primero hay que añadir el marco a la forma y después se incluyen los controles en el marco.

Fíjese en la figura anterior y añada a la forma *Panel de Control*, los controles que en ella se muestran y que acabamos de comentar, especificando las propiedades que se indican en la tabla siguiente.

	Caption	CtlName	Min	Max	Small Change	Large Change
Etiqueta 1	Título de la aplicación					
Caja de texto		NuevoTítulo				
Marco 1	Tamaño	Tamaño				
Etiqueta 2	Alto					
Etiqueta 3	Ancho					
Barra V. 1		Alto	2000	5000	57	570
Barra V. 2		Ancho	3000	8000	57	570
Marco 2	Color	Color				
Etiqueta 4	Rojo					
Etiqueta 5	Verde					
Etiqueta 6	Azul					
Barra H. 1		Rojo	0	255	1	20
Barra H. 2		Verde	0	255	1	20
Barra H. 3		Azul	0	255	1	20
Marco 3	Muestra	Muestra				
Botón 1	Aceptar	Aceptar				
Botón 2	Cancelar	Cancelar				

Ponga la propiedad **Text** de la caja de texto a valor nulo.

A continuación, uniremos el código a los controles y a la forma. Empecemos por la orden *Panel de Control* del menú *Fichero*. Cuando el usuario haga clic en

esta orden tiene que visualizarse el panel de control, el cual tiene quedar automáticamente inicializado con las propiedades actuales de la ventana principal.

```
Sub FicheroPanel_Click ()
  PanelDeControl.Show
End Sub
```

Cuando se carga una forma se da el suceso **Load** asociado con la misma, circunstancia que aprovecharemos para realizar las inicializaciones correspondientes. El procedimiento asociado con este suceso, se ejecuta solamente la primera vez que se visualiza el panel de control, ya que como veremos mas adelante, los botones *Aceptar* y *Cancelar* ocultan la forman, no la descargan. Las siguientes veces que se visualice el panel de control, conservará los valores que el usuario fijó la última vez.

```
Sub Form_Load ()
  Dim Color As Long
  Dim Rojo As Integer, Verde As Integer, Azul As Integer
  'Inicializar caja de texto
  PanelDeControl.NuevoTítulo.Text = Form1.Caption
  'Inicializar barras de tamaño
  PanelDeControl.Alto.Value = Form1.Height
  PanelDeControl.Ancho.Value = Form1.width
  'Inicializar la muestra de color
  PanelDeControl.Muestra.BackColor = Form1.BackColor
  'Inicializar barras de color
  Color = Form1.Point(0, 0)
  Rojo = Color And &HFF&
  Color = Color \ 256
  Verde = Color And &HFF&
  Color = Color \ 256
  Azul = Color And &HFF&
  PanelDeControl.Rojo.Value = Rojo
  PanelDeControl.Verde.Value = Verde
  PanelDeControl.Azul.Value = Azul
End Sub
```

Este procedimiento pone el nombre actual de la ventana principal en la caja de texto, inicializa la propiedad **Value** de cada una de las barras *Alto* y *Ancho* con el alto y ancho actuales de la ventana, inicializa el color de fondo del marco *Muestra* al color de fondo de la ventana principal, e inicializa la propiedad **Value** de cada una de las barras *Rojo*, *Verde* y *Azul* con los niveles de rojo, verde y azul actuales. La operación **And** de *Color* con *FF* nos da el byte menos significativo y la división entera de *Color* entre *256* realiza un desplazamiento a la derecha de un byt

Si el usuario quiere introducir un nuevo título para la ventana principal lo escribirá en la caja de texto. Si quiere modificar el tamaño de la ventana principal, actuará sobre las barras *Alto* y *Ancho*. Si quiere modificar el color de fondo de la ventana principal actuará sobre las barras *Rojo*, *Verde* y *Azul* y al mismo tiempo tiene que estar viendo en el marco *Muestra* las variaciones de color que está produciendo.

Cada vez que se desplaza el cursor a lo largo de una barra, se da el suceso **Change**. Por lo tanto, al fijar el nivel de color en las barras *Rojo*, *Verde* y *Azul*, los procedimientos invocados tienen que ejecutar la función **RGB** para esos valores, con el fin de actualizar el color de fondo del marco *Muestra*.

```
Sub Rojo_Change ()
  Muestra.BackColor = RGB(Rojo.Value, Verde.Value, Azul.Value)
End Sub

Sub Verde_Change ()
  Muestra.BackColor = RGB(Rojo.Value, Verde.Value, Azul.Value)
End Sub

Sub Azul_Change ()
  Muestra.BackColor = RGB(Rojo.Value, Verde.Value, Azul.Value)
End Sub
```

Una vez fijadas las características deseadas, el usuario pulsará el botón **Aceptar**, momento en el que se trasladan dichas características a la ventana principal y se oculta la forma. El procedimiento asociado con este botón [illegible]esenta a continuación.

```
Sub Aceptar_Click ()
  Form1.Caption = NuevoTítulo.Text
  Form1.Height = Alto.Value
  Form1.Width = Ancho.Value
  Form1.BackColor = PanelDeControl.Muestra.BackColor
  PanelDeControl.Hide
End Sub
```

Si en lugar de pulsar el botón *Aceptar* el usuario pulsa el botón *Cancelar*, no se tienen en cuenta las características fijadas y se oculta la forma.

```
Sub Cancelar_Click ()
  PanelDeControl.Hide
End Sub
```

Finalmente, cuando el usuario haga clic en la orden *Ejecutar*, [illegible] que visualizarse una caja de diálogo que solicite el nombre de una aplic[illegible] que desee

CAPÍTULO 3

ELEMENTOS DEL LENGUAJE

INTRODUCCIÓN

El lenguaje Visual Basic está muy cerca en muchos aspectos a sus predecesores QuickBasic, QBasic, y al Basic profesional, todos ellos de la firma Microsoft. Por ello, si no tiene conocimientos de alguno de estos lenguajes, le sugiero adquiera mi libro sobre *QuickBasic* o mi otro libro sobre *QBasic y MS-DOS* publicados por la editorial *RA-MA*. No obstante, este capítulo explica algunos mecanismos del lenguaje para que usted recuerde y pueda continuar leyendo esta obra con comodidad.

Visual Basic adopta los tipos de datos, estructuras, y muchas de las reglas sintácticas de sus predecesores. Sin embargo, hay algunas diferencias, como las que se indican a continuación, que tendrá que aprender.

- El ámbito de trabajo de las variables ahora es diferente.
- Las variables de tipo **Currency** contienen cantidades fraccionarias.
- Las llamadas a funciones (**Function**) deben incluir paréntesis, aunque no tengan argumentos.

Por supuesto, hay otras diferencias entre Visual Basic y QuickBasic las cuales se verán mas adelante.

COMENTARIOS

Cuando una frase va precedida de una comilla simple ('), Visual Basic interpreta que esa frase es un comentario y no ejecuta acción alguna sobre ella. Por ejemplo,

```
' Cálculo de la velocidad media
suma = 0    ' Se inicializa la variable suma al valor 0
```

CONSTANTES NUMÉRICAS Y DE CARACTERES

Una constante es un valor que no cambia durante la ejecución de un programa. Visual Basic admite números decimales (base 10), hexadecimales (base 16) y octales (base 8). Un número hexadecimal va precedido por **&H** y un número octal va precedido por **&O**. El siguiente ejemplo muestra los mismos números en decimal, hexadecimal y octal.

```
9       15      1034        son números decimales
&H9     &HF     &H40A       son números hexadecimales
&O11    &O17    &O2012      son números octales
```

Una constante de caracteres o constante alfanumérica es una cadena de caracteres encerrada entre comillas dobles. Por ejemplo,

```
"Grados centígrados:"
```

VARIABLES

Una variable contiene un valor que puede modificarse a lo largo de la ejecución de la aplicación. Cada variable tiene atributos propios como:

- *Nombre*. Es el nombre que utilizamos para referirnos a la variable en la aplicación.
- *Tipo*. El tipo determina qué clase de valores puede almacenar la variable.
- *Ámbito*. El ámbito de una variable especifica en qué parte de la aplicación la variable es conocida y por lo tanto puede utilizarse.

Nombres de variables

El nombre de una variable tiene que comenzar por una letra y puede tener hasta 40 caracteres de longitud.

Los caracteres pueden ser letras, dígitos, el carácter de subrayado y los caracteres de declaración del tipo de la variable (%, &, !, #, @, y $). El nombre de una va no puede ser una palabra reservada.

Una *palabra reservada* tiene un significado especial para Visual Basic. Son palabras reservadas las sentencias predefinidas (**For**) y los nombres de funciones (**Val**), métodos (**Hide**), propiedades (**Caption**) y operadores (**And**).

Tipos de datos

Una variable puede ser de alguno de los seis tipos siguientes:

Tipo	Descripción	Carácter de declaración del tipo	Rango
Integer	Entero (2 bytes)	%	-32768 a 32767
Long	Entero largo (4 bytes)	&	-2147483648 a 2147483647
Single (por defecto)	Real simple precisión (4 bytes)	!	-3.37E+38 a 3.37E+38
Double	Real doble precisión (8 bytes)	#	-1.67D+308 a 1.67D+308
Currency	Número con punto decimal fijo	@	-9.22E+14 a 9.22E+14
String	Cadena de caracteres	$	

Antes de utilizar una variable, hay que declarar su tipo. Una forma de hacer ésto es utilizando la sentencia **Dim** (o una de las palabras **Global** o **Static**). Cualquier declaración de éstas inicializa las variables numéricas con el valor cero y las variables alfanuméricas con el carácter nulo. Por ejemplo,

```
Dim I As Integer
Dim R As Double
Dim Nombre As String
Dim Etiqueta As String * 10
Dim F As Currency
Dim L As Long, X As Currency
```

Las sentencias anteriores declaran *I* como una variable entera, *R* como una variable real de precisión doble, *Nombre* como una variable para contener una cadena de caracteres de longitud variable, *Etiqueta* como una cade caracteres de longitud fija (10 caracteres), *F* como una variable fraccion como una

variable entera larga, y *X* como una variable fraccionaria. Observe que en una sentencia **Dim** puede realizar más de una declaración.

Otra forma de declarar una variable es utilizando los caracteres de declaración de tipo. Por ejemplo,

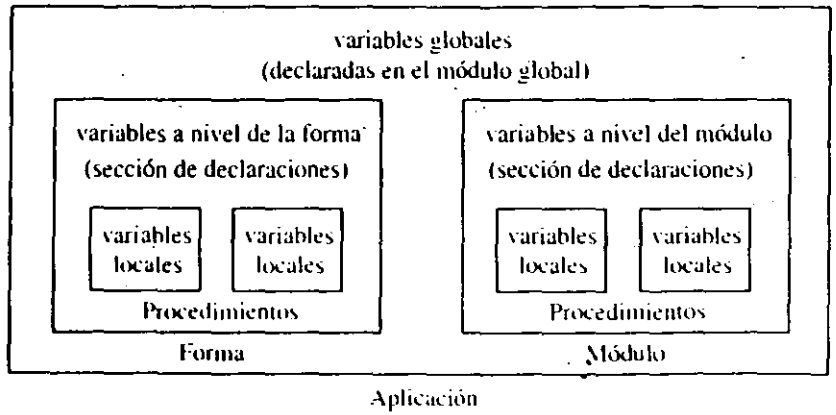
```
I%        Variable entera
R#        Variable real de precisión doble
Nombres   Cadena de caracteres
F@        Variable fraccionaria
```

Si una variable se utiliza y no se declara se asume que es de tipo **Single**.

Si de una variable se sabe que nunca va a contener un valor fraccionario, es mejor declararla como entera, ya que las operaciones con enteros son más rápidas. En caso contrario, si el valor no va a tener más de 4 dígitos decimales y no más de 14 dígitos enteros, es conveniente declararla como fraccionaria (**Currency**). En las variables de tipo **Currency** no tiene lugar el error producido en la conversión entre las bases 2 y 10, que sí tiene lugar cuando la variable es de tipo **Single** o **Double**.

Cuando una variable numérica de un tipo se asigna a otra variable numérica de un tipo diferente, Visual Basic realiza la conversión correspondiente.

Ámbito de las variables



Se entiende por ámbito de una variable, el espacio de la aplicación donde la variable es visible y por lo tanto se puede utilizar. La figura anterior indica los cuatro lugares donde se puede declarar una variable.

Variables locales

Una *variable local* se reconoce solamente en el procedimiento en el que está definida. Fuera de ese procedimiento la variable no es conocida. Su utilización más común es intervenir en cálculos intermedios.

Para declarar una variable local a un procedimiento, coloque la sentencia **Dim** correspondiente dentro del mismo. Por ejemplo,

```
Sub Grados_C_KeyPress (KeyAscii As Integer)
  Dim GradosFahr As Double
  If (KeyAscii = 13) Then
    GradosFahr = Val(Grados_C.Text) * 9 / 5 + 32
    Grados_F.Text = Format$(GradosFahr)
  End If
End Sub
```

Una variable local es reinicializada cada vez que se entra en el procedimiento. En otras palabras, una variable local no conserva su valor entre una llamada al procedimiento y la siguiente. Para hacer que esto suceda hay que declarar la variable estática. Visual Basic reinicializa una variable estática solamente la primera vez que se llama al procedimiento. Para declarar una variable estática utilice la palabra clave **Static** en lugar de **Dim**. Por ejemplo,

```
Static var_ent As Integer
```

Variables a nivel de la forma y a nivel del módulo

Una *variable declarada a nivel de la forma* puede ser compartida por todos los procedimientos de esa forma y una *variable declarada a nivel del módulo* puede ser compartida por todos los procedimientos de ese módulo. Una variable a nivel de la forma o del módulo hay que declararla en la sección de declaraciones de la forma o del módulo. Para editar esta sección, hay que abrir la ventana de código para un objeto y seleccionar "(general)" de la caja de objetos y "(declarations)" de la caja de sucesos para ese objeto. La forma de crear un nuevo módulo es ejecutando la orden **New Module** del menú **File**.

Este tipo de variables son por defecto *estáticas*.

Variables globales

Una *variable global* puede ser accedida desde cualquier parte de la aplicación. Para hacer que una variable sea global, hay que declararla en el *módulo global* de la aplicación. Este módulo no admite código, solo admite declaraciones. El nombre dado a este módulo por defecto es *Global.bas*. Para editarlo, selecciónelo en la ventana *Project* y haga clic en el botón *View Code*.

Para declarar en el módulo *Global.bas* una variable global, utilice la palabra clave **Global** en lugar de **Dim**. Por ejemplo,

```
Global var1_global As Double, var2_global As String
```

Variables con el mismo nombre en diferentes niveles

Una variable local, otra a nivel de la forma o del módulo y otra global pueden tener el mismo nombre, pero no son la misma variable. La regla para estos casos es que el procedimiento siempre utiliza la variable de nivel mas cercano (local, forma o módulo y global).

Si una variable aparece en un procedimiento y no está explícitamente declarada es por defecto local. Para asegurarse de que la variable es local, es mejor declararla explícitamente.

OPERADORES

La tabla que se muestra a continuación presenta el conjunto de operadores que soporta Visual Basic colocados de mayor a menor prioridad. Los operadores que aparecen sobre una misma línea tienen igual prioridad. Las operaciones entre paréntesis se evalúan primero, ejecutándose primero los paréntesis más internos.

Tipo	Operación	Operador
Aritmético	Exponenciación	^
	Cambio de signo	–
	Multiplicación y división	*, /
	División entera	\
	Resto de una división entera	**Mod**
	Suma y resta	+, –
Relac	Igual, distinto, mayor que, ...	=, <>, >, >=, <, <=

Tipo	Operación	Operador
Lógico	Negación	**Not**
(manejo de bits)	And	**And**
	Or inclusiva	**Or**
	Or exclusiva	**Xor**
	Equivalencia (opuesto a Xor)	**Eqv**
	Implicación	**Imp**
	(verdad si primer operando falso y segundo operando verdadero)	

SENTENCIAS

Una sentencia es una línea de texto que indica una o más operaciones a realizar. Una línea puede tener varias sentencias separadas unas de otras por dos puntos.

```
total = cantidad * precio: suma = suma + total
```

La sentencia más común en Visual Basic es la sentencia de asignación. Su forma general es,

variable = expresión

la cual indica que el valor que resulte de evaluar la *expresión* tiene que ser almacenado en la *variable* especificada. Si la expresión es numérica la variable tiene que ser también numérica y si la expresión es alfanumérica la variable tiene que ser también alfanumérica. Por ejemplo,

```
c = c + 1
Intereses = Capital * TantoPorCiento / 100
Mensaje = "La operación es correcta"
```

PROPIEDADES

Recordar que un objeto (forma o control) tiene asociadas varias propiedades. Para referirse a una propiedad de un objeto se utiliza la forma,

objeto.propiedad

Por ejemplo, supongamos el objeto *Texto* y su propiedad *Text*. Las siguientes operaciones serían válidas

```
Suma = Val(Texto.Text)
Texto.Text = "El total es " + Str$(Total)
```

La función **Val** convierte una cadena de caracteres en un número y la función **Str$** convierte un número en una cadena de caracteres. El operador + con cadenas de caracteres, enlaza las dos cadenas para formar una sola.

SENTENCIAS DE CONTROL

Las sentencias de control, denominadas también *estructuras de control*, permiten tomar decisiones y realizar un proceso repetidas veces. Visual Basic dispone de las siguientes estructuras, idénticas a las de sus predecesores como QuickBasic:

- **If ... Then ... Else**
- **Select Case**
- **For ... Next**
- **Do ... Loop**

If ... Then ... Else ...

Esta estructura permite ejecutar condicionalmente una o más sentencias y puede escribirse de las dos formas siguientes:

If *condición* **Then** *sentencia(s)* [**Else** *sentencia(s)*]

If *condición* **Then**
  sentencia(s)
[**Else**
  sentencia(s)]
End If

Si la *condición* es verdadera se ejecutan las sentencias que están a continuación de **Then** y si la condición es falsa se ejecutan las sentencias que están a continuación de **Else**, si esta cláusula ha sido especificada (una expresión entre corchetes - [ ] - es opcional). Por ejemplo,

```
If KeyAscii = 13 Then
  GradosFahr = Val(Grados_C.Text) * 9 / 5 + 32
  Grados_F.Text = Format$(GradosFahr)
End If
```


La condición es una expresión de **Boole**. Una expresión de Boole da un resultado verdadero o falso. Los operadores que intervienen en una expresión de Boole pueden ser de relación (=, <>, <, <=, >, y >=) y lógicos (**And**, **Or**, **Xor**, **Eqv**, **Imp**, y **Not**). Por ejemplo,

```
If A > B Or (A >= 1 And A <= 5) Then
```

Para indicar que se quiere ejecutar un bloque de sentencias de varios bloques dependientes cada uno de ellos de una condición, la estructura adecuada es la siguiente:

If *condición-1* **Then**
  sentencias-1
[**ElseIf** *condición-2* **Then**
  sentencias-2] ...
[**Else**
  sentencias-n]
End If

Si se cumple la *condición-1* se ejecutan las *sentencias-1*, y si no se cumple, se examinan secuencialmente las condiciones siguientes hasta **Else**, ejecutándose las sentencias correspondientes al primer **ElseIf** cuya condición se cumpla. Si todas las condiciones son falsas, se ejecutan las *sentencias-n* correspondientes a **Else**. En cualquier caso, se continúa en la sentencia que sigue a **End If**. Por ejemplo,

```
If C > 100 Then
  Resu = C * P * 0.6
ElseIf C >= 25 Then
  Resu = C * P * 0.8
ElseIf C >= 10 Then
  Resu = C * P * 0.9
Else
  Resu = C * P
End If
```

Select Case

Esta sentencia permite ejecutar una de varias acciones en función del valor de una expresión.

Select Case *expr-test*
Case *lista 1*
  [*sentencias 1*]
[**Case** *lista 2*

[*sentencias 2*]] . . .
[**Case Else**
[*sentencias n*]]
End Select

donde *expr-test* es una expresión numérica o alfanumérica, y *lista i* es una lista que puede tener cualquiera de las formas siguientes:

expresión[, *expresión*] . . .
expresión **To** *expresión*
Is *operador-de-relación expresión*
combinación de las anteriores separadas por comas

Aquí *expresión* es cualquier expresión numérica o de caracteres del mismo tipo que *expr-test*. Por ejemplo:

```
Case Is < x          'expr-test < x
Case 3               'expr-test = 3
Case x To 20         'expr-test = x, x-1, ...,20
Case 3, x            'expr-test = 3, x
Case -1, x To 5      'expr-test = -1, x, x+1, ...,5
Case "si", "SI"      'expr-test = "si", "SI"
Case Is >= 10        'expr-test >= 10
```

Cuando se utiliza la forma *expresión* **To** *expresión*, el valor más pequeño debe aparecer en primer lugar.

Cuando se ejecuta una sentencia **Select Case**, Visual Basic evalúa la *expr-test* y busca el primer **Case** que incluya el valor evaluado, ejecutando a continuación el correspondiente bloque de sentencias. Si no existe un valor igual a la *expr-test*, entonces se ejecutan las sentencias a continuación de **Case Else**. En cualquier caso, el control pasa a la siguiente sentencia a **End Select**. Por ejemplo.

```
Select Case X
  Case 1
    Text1.Text = "1"
  Case 2, 3
    Text1.Text = "2 o 3"
  Case 4 To 9
    Text1.Text = "4 a 9"
  Case Else
    Text1.Text = "X< > 0-9"
End Select
```


For ... Next

La sentencia **For** da lugar a un lazo o bucle y permite ejecutar un conjunto de sentencias, cierto número de veces.

For *variable* = *expresión 1* **To** *expresión 2* [**Step** *expresión 3*]
[*sentencias*]
[**Exit For**]
[*sentencias*]
Next [*variable* [,*variable*]. . .]

Cuando se ejecuta una sentencia **For**, primero se asigna el valor de la *expresión 1* a la *variable* y se comprueba, cuando el valor de la *expresión 3* es positivo, si la *variable* es mayor que la *expresión 2*, en cuyo caso se salta el cuerpo del bucle y se continúa en la línea a continuación de **Next**. En otro caso, se ejecutan las líneas de programa que haya a continuación de la sentencia **For** hasta que aparezca la sentencia **Next**. Entonces la *variable* se incrementa en el valor de la *expresión 3* o en *1* si **Step** no se especifica, volviéndose a efectuar la comparación entre la *variable* y la *expresión 2*, y así sucesivamente.

La sentencia **Exit For** permite salir de un bucle **For** ... **Next** antes de que éste finalice.

Un bucle **For** ... **Next** se ejecuta más rápidamente cuando la *variable* es entera, y las *expresiones 1, 2, 3*, constantes.

Ejemplo:

```
Sub Form_Click ()
  Dim I As Integer, Suma As Integer
  For I = 1 To 99 Step 2   ' Para I=1,3,5,... hasta 99
    Suma = Suma + I
  Next I
  Print Suma
End Sub
```

Este ejemplo indica que cuando hagamos clic sobre la forma, se visualizará sobre la misma la suma de los números impares entre 1 y 99.

Si el valor de la *expresión 3* es negativo, se comprueba si la *variable* es menor que la *expresión 2*, en cuyo caso se pasa a ejecutar la siguiente sentencia a

Next. En otro caso, se ejecuta el bucle y se decrementa la *variable*. Este proceso se repite hasta que la *variable* sea menor que la *expresión 2*.

Ejemplo:

```
Sub Form_Click ()
  Dim I As Integer, Suma As Integer
  For I = 99 To 1 Step -2  ' Para I=99,97,... hasta 1
    Suma = Suma + I
  Next I
  Print Suma
End Sub
```

Este ejemplo conduce al mismo resultado que el anterior.

Do ... Loop

Un **Loop** (bucle) repite la ejecución de un conjunto de sentencias mientras una condición dada sea cierta, o hasta que una condición dada sea cierta. La condición puede ser verificada antes o después de ejecutarse el conjunto de sentencias.

Formato 1:

Do [{**While/Until**} *condición*]
  [*sentencias*]
  [**Exit Do**]
  [*sentencias*]
Loop

Formato 2:

Do
  [*sentencias*]
  [**Exit Do**]
  [*sentencias*]
Loop [{**While/Until**} *condición*]

donde *condición* es cualquier expresión numérica, relacional o lógica.

Esta sentencia nos permite realizar varias estructuras diferentes. Permite, como se aprecia en los formatos, crear bucles con la condición de terminación al final o al principio del bloque de sentencias. Por ejemplo.


```
Sub Form_Click ()
  Dim I As Integer, Suma As Integer
  I = 1
  Do While I <= 99   ' Hacer mientras I <= 99
    Suma = Suma + I
    I = I + 2
  Loop
  Print Suma
End Sub
```

Este ejemplo indica que cuando hagamos clic sobre la forma, se visualizará sobre la misma la suma de los números impares entre 1 y 99. El mismo resultado se obtiene con el ejemplo que se presenta a continuación.

```
Sub Form_Click ()
  Dim I As Integer, Suma As Integer
  I = 99
  Do                  ' Hacer hasta que I < 1
    Suma = Suma + I
    I = I - 2
  Loop Until I < 1
  Print Suma
End Sub
```

La sentencia **Exit Do** permite salir de un bucle **Do ... Loop** antes de que finalice éste.

PROCEDIMIENTOS Y FUNCIONES

La base de una aplicación en Visual Basic la forman sus procedimientos conducidos por sucesos. Un *procedimiento conducido por un suceso* es el código que es invocado cuando un objeto reconoce que ha ocurrido un determinado suceso.

También podemos crear procedimientos generales. Un *procedimiento general* es invocado cuando se hace una llamada explícita al mismo, desde cualquier parte de la aplicación. Una buena razón para utilizar este tipo de procedimientos, es eliminar la necesidad de duplicar código cuando varios procedimientos conducidos por sucesos necesiten ejecutar un mismo proceso. Por ejemplo, visualizar un diagrama de barras. En este caso, el código común se coloca en un procedimiento general que será invocado desde cada procedimiento conducido por un suceso, que necesite ejecutar dicho proceso.

Un procedimiento general puede escribirse como procedimiento **Sub** o como función **Function**. Un procedimiento conducido por un suceso siempre es un procedimiento **Sub**.

Ámbito de un procedimiento

Cuando un procedimiento es llamado para su ejecución, Visual Basic busca ese procedimiento en la forma o módulo actual. Si no lo encuentra, entonces continúa la búsqueda en el resto de los módulos de la aplicación.

Consecuentemente, un procedimiento definido en un módulo puede invocarse desde cualquier parte de la aplicación, pero un procedimiento ligado a una forma solo puede ser llamado desde otros procedimientos ligados también a esa forma. Por lo tanto, el nombre para un procedimiento definido en un módulo debe ser único para todos los módulos, mientras que el nombre para un procedimiento definido en una forma puede utilizarse también en otra forma.

Crear un procedimiento general

Para crear un procedimiento general, primero se abre la ventana de código correspondiente a la forma o al módulo y a continuación se ejecuta la orden **New Procedure** del menú **Code**.

Para editar un procedimiento general existente, seleccione "(general)" de la caja de objetos en la ventana de código y a continuación seleccione el procedimiento de la caja de procedimientos.

Funciones (Function)

La sintaxis correspondiente a una función es la siguiente:

[Static] Function *nombre* ([*parámetros*]) [**As** *tipo*]
  [*sentencias*]
  [*nombre* = *expresión*]
  [Exit Function]
  [*sentencias*]
  [*nombre* = *expresión*]
End Function

nombre — es el nombre de la función y tipo de datos que devuelve, especificado por un sufijo de tipo de datos (%, &, !, #, @, o $). El tipo de datos devuelto por la función también puede indicarse utilizando la cláusula **As** *tipo* (**Integer**, **Long**, **Single**, **Double**, **Currency** o **String**)

parámetros — son una lista de identificadores separados por comas que representan variables. Cuando se llama la función, Visual Basic asigna el valor de cada argumento en la llamada al parámetro correspondiente de la función.

expresión — define el valor devuelto por la función. Este valor es almacenado en el propio *nombre* de la función, que actúa como variable dentro del cuerpo de la misma. Si no se efectúa esta asignación, el resultado devuelto será 0 si ésta es numérica, o nulo ("") si la función es de caracteres.

Exit Function — permite salir de una función. **Exit Function** no es necesaria a no ser que se necesite retornar a la sentencia inmediatamente a continuación de la que efectuó la llamada antes de que la función finalice.

End Function — esta sentencia, al igual que **Exit Function**, devuelve el control a la sentencia inmediatamente a continuación de la que efectuó la llamada, continuando de esta forma la ejecución del programa.

La llamada a una función es de la forma:

variable = *nombre* ([*argumentos*])

argumentos — son una lista de constantes, variables o expresiones separadas por comas. El número de argumentos debe ser igual al número de parámetros de la función. Los tipos de los argumentos deben coincidir con los tipos de sus correspondientes parámetros.

En cada llamada a una función hay que escribir los paréntesis, aunque ésta no tenga argumentos. Una llamada a una función es parte de una expresión.

El siguiente ejemplo corresponde a una función que devuelve como resultado el factorial de un número N.

```
Function Factorial (N As Integer) As Long
  Dim F As Long
  If N = 0 Then
    Factorial = 1
  Else
    F = 1
```

```
    Do While N > 0
      F = N * F
      N = N - 1
    Loop
    Factorial = F
  End If
End Function
```

La llamada a esta función se hará de la forma siguiente:

```
Fact = Factorial(Num)
```

Procedimientos (Sub)

La sintaxis que define un procedimiento es la siguiente:

[Static] Sub *nombre*[*(parámetros)*]
  [*sentencias*]
  [Exit Sub]
  [*sentencias*]
End Sub

La explicación es análoga a la dada para funciones **(Function)**.

La llamada a un procedimiento puede ser de alguna de las dos formas siguientes:

Call *nombre* [*(argumentos)*]
nombre [*argumentos*]

A diferencia de una función, un procedimiento no puede ser utilizado en una expresión.

Para declarar procedimientos externos, esto es, procedimientos contenidos en una DLL, utilice la sentencia **Declare**. Una sentencia **Declare** puede aparecer en el módulo global o en la sección de declaraciones de la forma o del módulo.

El siguiente ejemplo corresponde a un procedimiento (Sub) que devuelve como resultado el factorial de un número N.

```
Sub Factorial (N As Integer, F As Long)
  If N = 0 Then
    F = 1
  Else
```



```
    F = 1
    Do While N > 1
      F = N * F
      N = N - 1
    Loop
  End If
End Sub
```

La llamada a este procedimiento puede ser de cualquiera de las dos formas siguientes:

```
Factorial N, F

Call Factorial(N, F)
```

Declarar todas las variables locales como estáticas

Para hacer que todas las variables locales de un procedimiento sean por defecto estáticas, hay que colocar al principio de la cabecera del procedimiento la palabra clave **Static**. Por ejemplo,

```
Static Sub Proc_1 (X As Double, N As Integer)
  ...
End Sub
```

Argumentos por referencia y por valor

En las funciones **(Function)** y en los procedimientos **(Sub)**, los argumentos se pasan por **referencia**: de este modo cualquier cambio de valor que sufra un parámetro en el cuerpo de la función o del procedimiento, también se produce en el argumento correspondiente de la llamada a la función o al procedimiento.

Cuando se llama a una función o a un procedimiento, se podrá especificar que el valor de un argumento no sea cambiado por la función o por el procedimiento, poniendo dicho argumento entre paréntesis, en la llamada. Un argumento entre paréntesis en la llamada es un argumento pasado por **valor**. Por ejemplo,

```
Factorial (Num), Fact
```

En la llamada a la función *Factorial* de este ejemplo, el argumento *Num* es pasado por valor.

Otra forma de especificar que un argumento será siempre pasado por valor es anteponiendo la palabra **ByVal** a la declaración del parámetro en la cabecera del procedimiento (Sub o Function). Por ejemplo,

```
Sub Factorial (ByVal N As Integer, F As Long
  ...
End Sub
```

Procedimientos recursivos

Se dice que una función **(Function)** es recursiva o que un procedimiento **(Sub)** es recursivo si se llaman a sí mismos.

La función **factorial**, cuyo programa se presenta a continuación, es recursiva.

```
Function Factorial (N As Integer) As Long
  If N = 0 Then
    Factorial = 1
  Else
    Factorial = N * Factorial(N - 1)
  End If
End Function
```

TIPOS DE DATOS Form y Control

Los tipos de datos **Form** y **Control** sólo pueden utilizarse en las declaraciones de los parámetros de un procedimiento **(Sub o Function)**.

Un parámetro declarado de tipo **Form** puede operar sobre cualquier número de formas. Por ejemplo, supongamos que queremos inicializar algunas de las formas de nuestra aplicación, con el mismo color de fondo y del primer plano. Esto puede hacerse escribiendo un procedimiento general al que se le pueda pasar como argumento un objeto, en este caso una forma. Esto no sería posible sin el tipo **Form**.

```
Sub ColorForma (FormaN As Form)
  FormaN.BackColor = &HFFFFFF
  FormaN.ForeColor = &H0&
End Sub
```

Un parámetro declarado de tipo **Control** puede operar sobre cualquier número de controles. Por ejemplo, supongamos que queremos inicializar algunos de los colores de las formas de nuestra aplicación, con el mismo color de fondo y

del primer plano. Esto puede hacerse escribiendo un procedimiento general al que se le pueda pasar como argumento un objeto, en este caso un control. Esto no sería posible sin el tipo **Control**.

```
Sub ColorControl (ControlN As Control)
  ControlN.BackColor = &HFFFFFF
  ControlN.ForeColor = &H0&
End Sub
```

Para tener acceso a estos procedimientos desde cualquier parte de la aplicación, los incluiremos en un módulo.

Cuando estos procedimientos sean invocados, recibirán como argumento el nombre de cualquier forma o control que queramos inicializar a los colores en ellos definidos. Por ejemplo, supongamos que el siguiente procedimiento pertenece a la forma *Form1*.

```
Sub Form_Load ()
  ColorForma Form1    'Pone color a la forma Form1
  ColorControl Text1  'Pone color al control Text1
  ColorControl Text2  'Pone color al control Text2
  Form2.Show          'Carga y visualiza la forma Form2
End Sub
```

El suceso **Load** ocurre cuando se carga una forma. Quiere esto decir que cuando se ejecute la aplicación, se ejecutará el procedimiento *Form_Load* que pone color a la forma *Form1* y a los controles especificados. También, como última acción, carga y visualiza la forma *Form2*. Por lo tanto, si suponemos que el siguiente procedimiento pertenece a la forma *Form2*, ésta y los controles especificados serán inicializados al color definido por los procedimientos llamados.

```
Sub Form_Load ()
  ColorForma Form2
  ColorControl Text1
End Sub
```

Tipo de un control pasado como argumento

Cuando se llama a un procedimiento al que se le pasa como argumento un control, se puede determinar el tipo exacto de control que se ha pasado utilizando la sentencia,

{If | ElseIf} TypeOf *control* **Is** *tipo-de-control* **Then**

donde *control* es el valor pasado y *tipo-de-control* es una palabra clave de las siguientes:

CheckBox	**Frame**	**PictureBox**
ComboBox	**HScrollBar**	**TextBox**
CommandButton	**Label**	**Timer**
DirListBox	**ListBox**	**VScrollBar**
DriveListBox	**Menu**	
FileListBox	**OptionButton**	

Por ejemplo, el siguiente fragmento coloca una cadena de caracteres en un control, si el control es una caja de texto.

```
Sub InicioControl (ControlN As Control)
  ...
  If TypeOf ControlN Is TextBox Then
    ControlN.Text = "Correcto"
  End If
  ...
End Sub
```

La cláusula **ElseIf** se utiliza igual que se indicó al hablar de la estructura **If** en este mismo capítulo. Por ejemplo,

```
Sub InicioControl (ControlN As Control)
  ...
  If C = 1 Then
    Exit Sub
  ElseIf TypeOf ControlN Is TextBox Then
    ControlN.Text = "Correcto"
  End If
  ...
End Sub
```

ARRAYS

Un array permite referirse a una serie de variables por un mismo nombre y referenciar una única variable de la serie utilizando un índice.

Visual Basic, igual que sus predecesores, permite definir *arrays de variables* de una o más dimensiones y de cualquier tipo de datos, e introduce una nueva

clase de arrays, *arrays de controles*, necesarios para escribir menús, para crear nuevos controles en tiempo de ejecución, o para hacer que una serie de controles tengan asociado un mismo procedimiento para cada tipo suceso.

Arrays de variables

Un array de este tipo tiene que declararse utilizando código y puede tener una o más dimensiones. Para declarar un array estático (con un número fijo de elementos), Visual Basic hace tres consideraciones importantes:

- Para declarar un array en el módulo global, utilice la sentencia **Global**.
- Para declarar un array a nivel de módulo o de la forma, utilice la sentencia **Dim**.
- Dentro de un procedimiento, utilice la sentencia **Static**.

Ejemplos:

```
Dim Array_A(19) As String
```

Este ejemplo declara un array de una dimensión, *Array_A*, con veinte elementos, *Array_A(0)*, *Array_A(1)*, ..., *Array_A(19)*, cada uno de los cuales permite almacenar una cadena de caracteres.

```
Dim Array_B(3, 1 To 6) As Integer
```

Este ejemplo declara un array de dos dimensiones, *Array_B*, con 4×6 elementos, *Array_B(0,1)*, ..., *Array_B(3,6)*, de tipo entero.

```
Static Array_C(1 To 5, 1 To 5) As Integer
```

Este ejemplo declara un array de dos dimensiones, *Array_C*, con 5×5 elementos, *Array_C(1,1)*, ..., *Array_C(5,5)*, de tipo entero.

```
Global Array_D(1 To 12) As String
```

Este ejemplo declara un array de una dimensión, *Array_D*, con doce elementos, *Array_D(1)*, ..., *Array_D(12)*, cada uno de los cuales permite almacenar una cadena de caracteres.

A diferencia de otras versiones de Basic, Visual Basic no permite declarar implícitamente un array. Un array tiene que ser declarado explícitamente.

Arrays dinámicos

Cuando las dimensiones de un array no son siempre las mismas, la mejor forma de especificarlas es mediante variables. Un array declarado de esta forma es un array dinámico. El espacio necesario para un array estático se asigna al iniciarse el programa y permanecerá fijo. El espacio para un array dinámico será asignado durante la ejecución del programa. Un array dinámico puede ser redimensionado en cualquier momento durante la ejecución.

Para crear un array dinámico:

1. Declare el array con una sentencia **Dim** o **Global**, y una lista de dimensiones vacía.
2. Asigne el número actual de elementos con la sentencia **ReDim**.

La sentencia **ReDim** puede aparecer solamente en un procedimiento y permite cambiar el número de elementos del array, no el número de dimensiones.

Cuando se crea un array que es local a un procedimiento, el primer paso se recomienda pero no es necesario. Al declarar un array dinámico primero con **Dim**, el número de dimensiones queda limitado a 8. Si son necesarias más dimensiones, hay que eliminar el primer paso y utilizar directamente la sentencia **ReDim**.

Por ejemplo, si declaramos el array *Array_A* a nivel de la forma,

```
Dim Array_A() As Integer
```

más tarde, un procedimiento *Cálculo* puede asignar espacio para el array, como se indica a continuación.

```
Sub Cálculo ()
  ...
  ReDim Array_A(F, C)
  ...
End Sub
```

Arrays de controles

Un array de controles se crea al asignar el mismo nombre a dos o más controles (p[illegible]d **CtlName**) durante el diseño de una aplicación, o bien, asignando un valo[illegible] propiedad **Index** de un control. La última forma genera un array de un solo elemento, lo cual es útil cuando se quiere crear controles en tiempo de ejecución.

A diferencia de un array ordinario, un array de controles no se declara en el código, está limitado a una sola dimensión, no tienen definido el límite superior (el límite inferior de un array de controles es cero), no puede ser pasado como un único argumento (cada elemento debe ser pasado como un argumento separado), y los índices no necesariamente tienen que ser consecutivos. Por ejemplo, *MiArray(1)* y *MiArray(7)* pueden ambos corresponder a controles, aunque no haya controles con los índices 2 a 6.

Cuando Visual Basic crea un array de controles, asigna por defecto los índices 0, 1, 2, ..., los cuales pueden modificarse a voluntad del usuario, cambiando el valor de la propiedad **Index**. Esta operación sólo puede realizarse durante el diseño.

Cuando cualquier control del array reconoce un suceso, Visual Basic pasa el índice como un argumento extra. Este argumento siempre va antes que cualquier otro argumento. Por ejemplo,

```
Sub MiArray_Click (Index As Integer, Dato As Integer)
```

El valor de *Index* puede utilizarse dentro del procedimiento para determinar qué control ha recibido el suceso. Un elemento del array de controles puede referenciarse escribiendo *NombreArray(Index)*, nomenclatura que puede utilizarse en cualquier lugar donde pueda especificarse el nombre del control.

EL MÓDULO GLOBAL

El módulo global contiene las declaraciones de variables, las definiciones de constantes, y el tipo de información que es reconocida por toda la aplicación.

Cada aplicación tiene un único módulo global, aunque no se utilice. El módulo global puede incluir:

- Declaraciones de variables globales. En las declaraciones de variables, **Dim** es sustituida por **Global**.

  ```
  Global Precio As Currency
  ```

- Sentencias **Def***tipo* (**DefInt**, **DefLng**, **DefSng**, **DefDbl**, **Def[illegible]** **DefStr**).

  ```
  DefInt H-K, N
  ```

- Definiciones de constantes globales. En la definición de una constante, **Global** tiene que preceder a **Const**.

```
Global Const TRUE = -1, FALSE = 0
```

- Tipos definidos por el usuario

```
Type Ficha
  Nombre As String
  Dni As Long
End Type
```

- Sentencias **Declare**

ESTRUCTURAS

Una estructura o registro es un nuevo tipo de datos que puede ser manipulado de la misma forma que los tipos predefinidos. Una estructura puede definirse como una colección de datos de diferentes tipos relacionados lógicamente.

Para crear una estructura hay que utilizar la sentencia **Type ... End Type**. Esta sentencia solamente puede aparecer en el módulo global. Por ejemplo,

```
Type Ficha
  Nombre As String
  Dirección As String * 40
  Teléfono As Long
  Dni As Long
End Type
```

Este ejemplo declara un tipo de estructura de datos denominado *Ficha* que consta de cuatro miembros o campos denominados *Nombre*, *Dirección*, *Teléfono* y *Dni*.

Una vez definido un tipo de estructura, podemos declarar variables de ese tipo. Por ejemplo:

```
Dim Alum As Ficha
```

Este ejemplo declara la variable *Alum* de tipo *Ficha*. Esta variable está formada por los miembros *Nombre*, *Dirección*, *Teléfono* y *Dni*.

Para referirse a un determinado miembro de una estructura se utiliza la notación *variable.miembro*. Por ejemplo,

```
Alum.Dni = 11133344
```

Un miembro de una estructura puede ser de un tipo definido por el usuario. Por ejemplo, observe a continuación el miembro *Alta*.

```
Type Fecha
  Día As Integer
  Mes As Integer
  Año As Integer
End Type

Type Ficha
  Alta As Fecha
  Nombre As String
  Dirección As String * 40
  Teléfono As Long
  Dni As Long
End Type
```

También, una estructura puede asignarse a otra estructura. Por ejemplo,

```
Dim Alum1 As Ficha, Alum2 As Ficha

Alum1.Alta.Día = 15
Alum2 = Alum1
```

CAPÍTULO 4

CONTROLES MÁS COMUNES

INTRODUCCIÓN

Los controles más comunes en una aplicación Windows son las cajas de texto, las etiquetas y los botones. Las cajas de texto son particularmente importantes porque permiten realizar la entrada de datos para una aplicación y visualizar los resultados producidos por la misma. Quiere esto decir que ellas sustituyen a las sentencias básicas de entrada/salida (E/S) del lenguaje Basic (INPUT, PRINT, INKEYS, etc.). Las etiquetas son cajas de texto no modificables por el usuario. Su finalidad es informar al usuario de qué tiene que hacer y para qué es cada control. Por último, un botón permite al usuario ejecutar una orden cuando sea preciso.

Los ejemplos que se presentan a continuación permitirán aprender a utilizar estos controles y otros, además de sus propiedades.

CONVERSIÓN DE TEMPERATURAS

La aplicación de conversión de grados centígrados a fahrenheit y viceversa requiere un medio de comunicación (interfaz) de forma que cuando el usuario introduzca en un área una temperatura en grados centígrados, en otra área se visualice la temperatura equivalente en grados fahrenheit y cuando en este otro área se introduzca una temperatura en grados fahrenheit, en la primera se visualice la temperatura correspondiente en grados centígrados.

Recuerde que una **interfaz** se define como el medio de comunicación entre el usuario y la aplicación.

DESARROLLO DE LA APLICACIÓN

Antes de crear una aplicación debemos responder algunas preguntas como las siguientes:

- ¿Qué objetos forman el medio de comunicación?
- ¿Qué sucesos hacen que el medio de comunicación responda?
- ¿Cuáles son los pasos a seguir para un desarrollo ordenado?

Objetos

La conversión de temperaturas implica los siguientes objetos:

- Una forma que permita implementar nuestro medio de comunicación.
- Una caja de texto para visualizar los grados centígrados.
- Una caja de texto para visualizar los grados fahrenheit.
- Dos etiquetas que informen al usuario de la información que contiene cada caja de texto.

Sucesos

En esta aplicación se quiere que cuando un usuario escriba una temperatura en una caja y pulse **Enter**, el contenido de la otra caja se actualice automáticamente, y viceversa. Por lo tanto, el suceso para cada una de las cajas de texto es pulsar la tecla **Enter**, que en Visual Basic se denomina **KeyPress**.

Pasos a seguir durante el desarrollo

1. Crear una nueva aplicación.
2. Mover y ajustar el tamaño por defecto de la forma.
3. Dibujar los controles.
4. Definir las propiedades de la forma y de los controles.
5. Escribir el código para cada uno de los objetos.
6. Guardar la aplicación.
7. [illegible]icar la aplicación.
8. C[illegible]an fichero ejecutable.

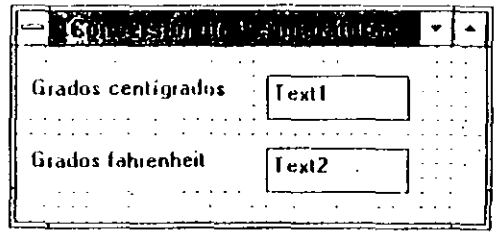
Estos pasos fueron expuestos con detalle anteriormente en el Capítulo 2. Por ello, se supone que los conceptos generales, cómo crear una nueva aplicación, mover y ajustar el tamaño por defecto de la forma y dibujar los controles, no presentan ya ninguna dificultad.

La forma, los controles y sus propiedades

Conocidos los objetos y los sucesos procedemos a dibujar el medio de comunicación. Para ello, creamos una nueva aplicación y cambiamos el título *Form1* de la forma que se visualiza, por el título *Conversión de temperaturas*. Esto se hace modificando el contenido de la propiedad **Caption** de la forma. A continuación ejecutamos la orden **Save Project** para guardar toda la parte de la aplicación que hemos realizado hasta ahora. Pongamos a la forma el nombre *Conver.frm* y a la aplicación o proyecto el nombre *Conver.mak* (las extensiones son añadidas automáticamente por Visual Basic). Es una buena idea salvar frecuentemente el trabajo que estamos realizando.

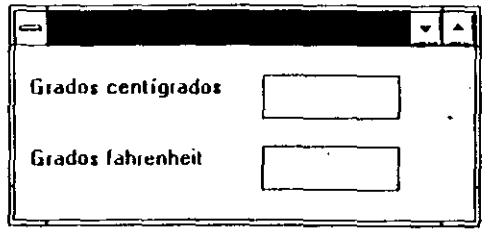
A continuación dibujamos los controles. Para dibujar una etiqueta haga doble clic sobre la utilidad "A" en la barra de utilidades y para crear una caja de texto haga doble clic sobre la utilidad "ab".

Dibujamos una etiqueta, la movemos al lugar deseado, cambiamos el título *Label1* por *Grados centígrados*, y ajustamos su tamaño. Después, dibujamos una caja de texto, la movemos a continuación de la etiqueta y cambiamos su nombre **(Ctl Name)** *Text1* por *Grados_C*. Siguiendo los mismos pasos, dibujamos una segunda etiqueta denominada *Grados fahrenheit* y otra caja de texto que denominaremos *Grados_F*. Una vez ajustado el lugar y el tamaño de los controles, ajustamos el tamaño de la forma. El medio de comunicación obtenido será similar al de la figura siguiente.



Una *etiqueta* tiene la característica de que no puede ser modificada por el usuario de la aplicación.

Para mejorar la apariencia inicial de nuestro medio de comunicación podemos realizar alguna cosa más. En primer lugar vamos a borrar los contenidos *Text1* y *Text2* de las cajas de texto. Para hacer ésto, seleccionamos la caja de texto primera y en la barra de propiedades ponemos la propiedad **Text**, que seleccionaremos de la caja de propiedades. A continuación modificamos el contenido de la misma borrando *Text1*. Para borrar *Text2* procederemos de forma similar. En segundo lugar eliminaremos la rejilla, para lo cual hay que ejecutar la orden **Grid Settings** del menú **Edit** y en la ventana de diálogo que se nos presenta, desactivamos la opción **Show Grid** (mostrar rejilla). Por último cambiaremos el color de fondo de las cajas de texto. Para ello, seleccionamos una a una cada caja de texto y su propiedad **BackColor**; después pulsamos el botón "..." que está a la derecha de la caja de edición, en la barra de propiedades, para visualizar el panel de colores; elegimos el color quinto de la tercera línea y cerramos esta ventana (**Alt+F4**). El resultado que se obtiene se muestra en la figura siguiente.



Ya tenemos la forma y los controles. Para hacerlos trabajar necesitamos unir a los mismos el código correspondiente.

Escribir el código

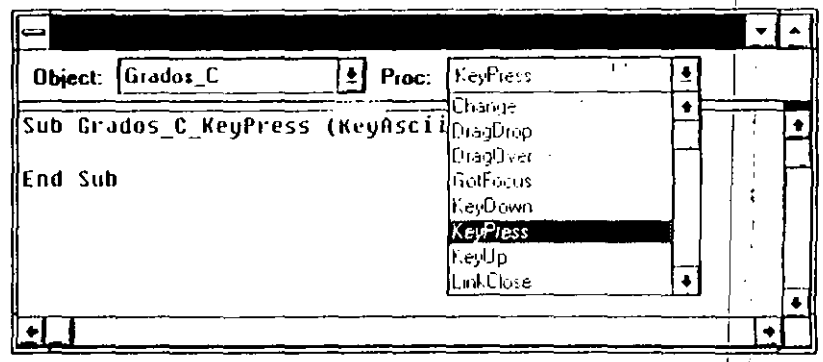
Para hacer que una aplicación responda a sucesos uniremos el código correspondiente para cada suceso, a cada uno de los controles. Este código asociado con cada control para un determinado suceso, recibe el nombre de *procedimiento del suceso*.

Para esta aplicación tenemos que escribir dos procedimientos, uno para cada control, ambos accionados por el suceso **KeyPress**.

El proceso que deseamos realizar es que cuando un usuario escriba una temperatura en una caja y pulse **Enter**, se actualice automáticamente el contenido de la otra caja con el valor resultante de la conversión correspondiente. Para realizar la conversión de una temperatura a otra utilizaremos las fórmulas siguientes:

```
GradosCent = (GradosFahr - 32) * 5 / 9
GradosFahr = (GradosCent * 9 / 5) + 32
```


En primer lugar desarrollaremos el procedimiento asociado con la caja de texto *Grados_C*, para el suceso **KeyPress**. La ventana de código correspondiente a este procedimiento, se visualiza haciendo un doble clic sobre esta caja de texto. Observemos la lista de sucesos (**Proc:**) que se pueden dar para que se ejecute este procedimiento, y elijamos **KeyPress**. Ahora la ventana de código presenta el esquema para el procedimiento *Grados_C_KeyPress*, como se aprecia en la figura siguiente.



Cuando en la caja *Grados_C* escribamos una cantidad (grados centígrados) y pulsemos **Enter** se ejecutará el procedimiento asociado *Grados_C_KeyPress*. Por lo tanto, este procedimiento tiene que convertir la cantidad escrita, en grados fahrenheit y almacenar el resultado en *Grados_F.Text*. Esta cantidad está almacenada como cadena de caracteres en la propiedad *Text* del objeto *Grados_C*, y como ya sabemos, se accede a ella utilizando la nomenclatura *Grados_C.Text*. Según lo expuesto el código resultante es el siguiente:

```
Sub Grados_C_KeyPress (KeyAscii As Integer)
  Dim GradosFahr As Double
  If (KeyAscii = 13) Then
    GradosFahr = Val(Grados_C.Text) * 9 / 5 + 32
    Grados_F.Text = Format$(GradosFahr)
  End If
End Sub
```

En segundo lugar desarrollaremos el procedimiento asociado con la caja de texto *Grados_F*, para el suceso **KeyPress**. La ventana de código correspondiente a este procedimiento, se visualiza haciendo un doble clic sobre esta caja de texto. Observemos la lista de sucesos (**Proc:**) que se pueden dar para que se ejecute este

procedimiento, y elijamos **KeyPress**. A continuación escribiremos en la ventana de código el procedimiento *Grados_F_KeyPress*, que se presenta a continuación.

```
Sub Grados_F_KeyPress (KeyAscii As Integer)
  Dim GradosCent As Double
  If (KeyAscii = 13) Then
    GradosCent = (Val(Grados_F.Text) - 32) * 5 / 9
    Grados_C.Text = Format$(GradosCent)
  End If
End Sub
```

Interceptando la tecla pulsada

Cada vez que el usuario pulsa una tecla, ocurren tres sucesos sobre el objeto actualmente seleccionado (objeto sobre el que está el cursor): **KeyDown**, **KeyPress**, y **KeyUp**. El suceso **KeyPress** es el más fácil de utilizar de los tres sucesos relacionados con el teclado. A diferencia de los sucesos **KeyDown** y **KeyUp**, el suceso **KeyPress** se genera solamente cuando se introduce un carácter ASCII. Esta definición excluye teclas especiales como teclas de función (**F1** a **F12**), teclas de movimiento del cursor (←, ↑, → y ↓) o la tecla **Del**.

El conjunto de caracteres ASCII es un subconjunto del conjunto de caracteres ANSI con el que trabaja Visual Basic. Los caracteres ASCII incluyen todos los caracterés imprimibles, las combinaciones **Ctrl+(A-Z)** y otros caracteres estándar como **Enter** (ASCII 13) y **BackSpace** (ASCII 8 - retroceso). Para interceptar cualquier otra tecla o combinación de teclas que no produzcan un código ASCII, se utilizarán los sucesos **KeyDown** y **KeyUp**.

El suceso **KeyPress** utiliza el argumento *KeyAscii*, el cual contiene el valor ASCII del carácter imprimible. Este valor puede ser convertido a un carácter utilizando la función **Chr$** y puede ser comparado utilizando los operadores de relación (=, <, >, <=, >= y <>).

Otras funciones relacionadas con caracteres son, **Asc(*cS*)** que devuelve el valor ASCII del carácter *cS* y **UCase$(*cS*)** que convierte el carácter *cS* a mayúsculas. Por ejemplo,

```
Sub [illegible] (KeyAscii As Integer)
  [illegible]                    ' Convierte a carácter
  [illegible]                    ' Convierte a mayúsculas
  [illegible]                    ' Convierte a valor ASCII
End Sub
```


El siguiente ejemplo demuestra como cancelar un carácter si no está dentro del rango permitido. Este ejemplo previene a la caja de texto de recibir cualquier carácter que no sea un número.

```
Sub Text1_KeyPress (KeyAscii As Integer)
  cS = Chr$(KeyAscii)                ' Convierte a carácter
  If cS < "0" Or cS > "9" Then       ' Si fuera de rango
    KeyAscii = 0                     ' Cancelar carácter
    Beep                             ' Pitido
  End If
End Sub
```

Los sucesos **KeyDown** y **KeyUp** utilizan los argumentos *KeyCode* y *Shift*. El primer argumento contiene el código ANSI (no ASCII) de la tecla pulsada y el segundo argumento contiene un valor 1, 2, o 4 dependiendo de que se haya pulsado la tecla **Shift**, **Ctrl**, o **Alt**, respectivamente. (si no se ha pulsado una de estas teclas el valor de *Shift* es 0) o un valor suma de los anteriores si se ha pulsado dos o más teclas simultáneamente (por ejemplo al pulsar **Shift+Alt** el argumento *Shift* toma el valor 5).

Los sucesos **KeyDown** y **KeyUp** no diferencian entre minúsculas y mayúsculas. Para hacer esta distinción tenemos que ver si el argumento *Shift* es 0 (minúscula) o 1 (mayúscula). Por ejemplo si pulsamos la tecla "a" el valor del argumento *KeyCode* es 65 y el valor del argumento *Shift* es 0; si pulsamos las teclas "Shift"+"a" el valor del argumento *KeyCode* es 65 y el valor del argumento *Shift* es 1. La razón de por qué estos sucesos son útiles está en el reconocimiento de teclas que no producen un código ASCII, como las teclas de función.

Una vez presentada la forma de proceder de estos sucesos, observemos cualquiera de los procedimientos anteriores, por ejemplo,

```
Sub Grados_C_KeyPress (KeyAscii As Integer)
  Dim GradosFahr As Double
  If (KeyAscii = 13) Then
    GradosFahr = Val(Grados_C.Text) * 9 / 5 + 32
    Grados_F.Text = Format$(GradosFahr)
  End If
End Sub
```

Cada vez que se pulse una tecla sobre la caja de texto *Grados_C*, se activa el procedimiento *Grados_C_KeyPress* y *KeyAscii* toma el valor correspondiente a la tecla pulsada. Por ello, para que la operación de conversión ocurra sólo en el momento de pulsar **Enter**, tenemos que verificar si la tecla pulsada ha sido **Enter** (ASCII 13). Esto es,

```
If (KeyAscii = 13) Then
  GradosFahr = Val(Grados_C.Text) * 9 / 5 + 32
  Grados_F.Text = Format$(GradosFahr)
End If
```

Ahora, guarde la aplicación (**Save Project**) y pruebe como funciona (**F5**).

CÓMO TRABAJA UN PROCEDIMIENTO

Como ejemplo, examinemos el código del procedimiento *Grados_C_KeyPress*. Muchos de los procedimientos que escribiremos a lo largo de esta obra son similares a este y trabajan de acuerdo con los siguientes puntos:

1. Se recupera el texto que el usuario escribe en las cajas de texto.
2. Se convierte el texto a un número para que pueda intervenir en los cálculos.
3. Se realizan los cálculos.
4. Se da formato a los resultados para visualizarlos en las cajas de texto.

Recuperando el texto de las cajas de texto

El contenido de cada caja de texto es representado por la propiedad **Text** y la forma de referenciar una propiedad de un objeto es *objeto.propiedad*. En nuestro ejemplo, *Grados_C.Text* representa el texto de la caja nombrada *Grados_C*.

Convirtiendo el texto a un número

El contenido de una caja de texto es una cadena de caracteres y por lo tanto, no tiene valor numérico. Quiere esto decir que si tenemos que realizar operaciones aritméticas con ese contenido, previamente hay que convertirlo a un número.

Para convertir una cadena de caracteres a un número, se utiliza la función **Val**.

Realizando cálculos

En el Capítulo 3 fueron expuestos los opetadores y el orden en el que se evalúan cuando en una expresión intervienen varios. Recuerde que una operación entre paréntesis tiene mayor prioridad y que las funciones se evalúan antes que cualquier otra operación. Por ejemplo,


```
GradosFahr = Val(Grados_C.Text) * 9 / 5 + 32
```

En este ejemplo, *Grados_C.Text* representa el texto de la caja nombrada *Grados_C*. La función **Val** convierte este texto a un número y después de realizar las operaciones indicadas, almacena el resultado en la variable *GradosFahr* de tipo **Double**.

Dando formato a la salida

Si queremos presentar en una caja de texto el valor numérico obtenido como resultado de una serie de operaciones aritméticas, la forma más fácil de hacerlo es utilizar la función **Str$** para convertirlo a una cadena de caracteres y asignar esta cadena a la propiedad **Text** de la caja de texto. Por ejemplo,

```
Grados_F.Text = Str$(GradosFahr)
```

La función **Format$** realiza la misma operación que **Str$** y además da formato a la salida de acuerdo al patrón especificado. Por ejemplo,

```
Grados_F.Text = Format$(GradosFahr, "#,##0.00")
```

El símbolo **#** representa un dígito cualquiera excepto los ceros no significativos. El símbolo **0** representa un dígito cualquiera incluyendo los ceros no significativos y además redondea el resultado. El punto indica la posición del punto decimal y la coma es el separador de los miles. Otros símbolos que también se pueden incluir son: **- + $ ( )** y **espacio**. Estos últimos se visualizan tal cual. El símbolo **%** presenta el valor por cien y añade al resultado ese símbolo.

Patrón	Entrada: 5	Entrada: -5	Entrada: .5
Sin patrón	5	-5	.5
0	5	-5	1
0.00	5.00	-5.00	0.50
#,##0	5	-5	1
#,##0.00	5.00	-5.00	0.50
#,###.##	5.	-5.	.5
0%	500%	-500%	50%
0.00%	500.00%	-500.00%	50.00%
0.00E+00	5.00E+00	-5.00E+00	5.00E-01
0.00E-00	5.00E0	-5.00E0	5.00E-01

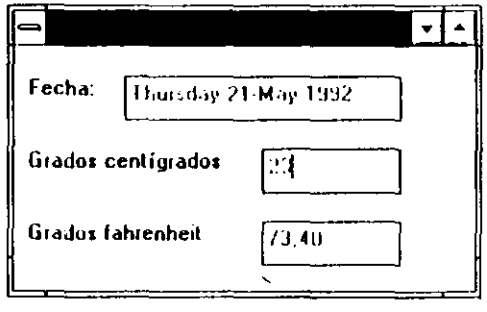
Si usted ha elegido en Windows como idioma el Español, observe como un punto decimal en la entrada o en el patrón, es convertido en una coma decimal. Igualmente la coma de los miles es convertida en el punto de los miles.

VISUALIZAR LA FECHA Y LA HORA

La función **Now** da como resultado la fecha y la hora actuales. Para visualizar esta fecha y hora bajo diversos patrones, utilice la función **Format$** con los símbolos especiales **d** (día), **m** (mes), **y** (año) y **h** (horas), **m** (minutos) y **s** (segundos). Por ejemplo,

Patrón	Format$(Now, "Patrón")
d-m-yy	21-5-92
dd/mm/yy	21/05/92
dd-mmm-yyyy	21-May-1992
d-mmm m-yyyy	21-May 5-1992
dddd dd - mmm m - yyyy	Thursday 21 - May 5 - 1992
h:m:s, d-mmm-yy	22:24:50, 21-May-92
hh:mm:ss AM/PM, dd-mmm-yyyy	10:24:50 PM, 21-May-1992

Como práctica, modifique la aplicación *Convert.mak* para que presente los resultados con dos decimales y también, visualice la fecha. El aspecto final será el indicado en la figura siguiente.



El proceso a seguir es el siguiente:

1. Abra la aplicación *Convert.mak* (**Open Project**).

2. Visualice la forma *Convert.frm*. Selecciónela en la ventana de la aplicación y haga clic en el botón **View Form**.
3. Visualice la ventana de código de *Grados_C* y añada formato al resultado.

```
Grados_F.Text = Format$(Grados_F, "#,##0.00")
```

4. Visualice la ventana de código de *Grados_F* y añada formato al resultado.

```
Grados_C.Text = Format$(Grados_C, "#,##0.00")
```

5. Ajuste el tamaño de la forma, añada los controles que se especifican en la siguiente tabla y modifique sus propiedades como se indica.

	Etiqueta	Caja de texto
Caption	Fecha:	(no tiene)
CtlName	Label3	Fecha
Text	(no tiene)	sin texto

6. Modifique el color de la caja de texto. Para ello, seleccione la caja de texto *Fecha* y su propiedad **BackColor**; después pulse el botón "..." que está a la derecha de la caja de edición, en la barra de propiedades, para visualizar el panel de colores; elija el color quinto de la tercera línea como color de fondo y negro para el texto. Cierre esta ventana (**Alt+F4**).

A continuación tenemos que añadir código para que se visualice la fecha. Queremos que cuando se ejecute la aplicación, la fecha aparezca en su caja de texto. Uno de los sucesos que ocurre cuando una aplicación se ejecuta, es el suceso **Load** asociado con la forma. Es aquí donde tenemos que escribir el código correspondiente para visualizar la fecha.

Haga un doble clic sobre cualquier parte exclusiva de la forma y le aparecerá la ventana de código correspondiente. Seleccione el suceso **Load** para este procedimiento y escriba el cuerpo del mismo como se indica a continuación.

```
Sub Form_Load ()
  Fecha.Text = Format$(Now, "dddd dd-mmm-yyyy")
End Sub
```

La función **Now** proporciona la fecha y **Format$** especifica el formato con el que se ha de visualizar. El resultado hay que almacenarlo en la propiedad **Text** del objeto *Fecha*, esto es, en *Fecha.Text*.

PASAR DE UN CONTROL A OTRO

Como es usual en Windows, el usuario puede pasar desde un control a otro pulsando la tecla **Tab** (⇥) o utilizando el ratón.

Esto quiere decir que el usuario, sobre la aplicación que hemos diseñado, puede situarse sobre la fecha y alterarla. Como esto no lo consideramos conveniente, tenemos que evitar que el usuario pulsando la tecla **Tab** pueda moverse a la caja de texto que visualiza la fecha. La forma de hacer esto es poniendo la propiedad **TabStop** a valor **False**.

Nosotros podemos hacer esto, seleccionando la propiedad **TabStop** de la caja de texto *Fecha* en la barra de propiedades y poniéndola a valor **False**.

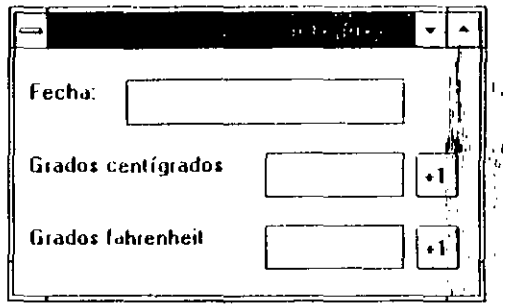
BOTONES DE ÓRDENES

Una forma de emitir una orden es pulsando un botón. Como es usual en Windows, el usuario puede pulsar un botón haciendo clic en él o seleccionándolo con la tecla **Tab** y pulsando la barra espaciadora (espacio en blanco). Estas acciones invocan al procedimiento conducido por el suceso **Click** del botón pulsado.

Cuando una forma tiene uno o más botones, uno y sólo uno de ellos puede ser "botón por defecto". Esta característica hace que cada vez que el usuario pulse **Enter**, se invoque el suceso **Click** de este botón, sin importar el control que esté enfocado. Cuando seleccionamos un control, si este es un botón, su título aparece rodeado por un borde; se dice entonces que "el botón está enfocado".

Para hacer que un botón sea el botón por defecto, hay que poner la propiedad **Default** del mismo a valor **True**. El botón por defecto se distingue de los demás, porque aparece rodeado con un borde más negro.

Como ejemplo, vamos a añadir a nuestra aplicación dos botones, uno para realizar un incremento de una unidad en la caja de texto *Grados_C* y otro para realizar un incremento de una unidad en la caja de texto *Grados_F*. El usuario podrá activar estos botones con el ratón o enfocándolos y pulsando la barra espaciadora. La figura siguiente muestra cual será el resultado después de hacer esta operación.



Para añadir uno de los botones haga doble clic en la utilidad botón del panel de utilidades, ajuste su tamaño, muévalo al lugar deseado, cambie el título por defecto por el título "+1" modificando su propiedad **Caption** y cambie su nombre de control "Command1" por "Orden1" modificando su propiedad **CtlName**. Para añadir el segundo botón siga los mismos pasos.

A continuación añadimos el código correspondiente para hacerles operativos. Lo que queremos que suceda es que al pulsar el botón, el contenido de la caja de texto correspondiente se incremente en una unidad. Por ejemplo, el contenido de la caja de texto *Grados_C* es *Grados_C.Text*. Este valor lo incrementamos en una unidad utilizando la sentencia,

```
Grados_C.Text = Format$(Val(Grados_C.Text) + 1)
```

Al cambiar el valor de la caja de texto *Grados_C* debe cambiar también el valor de la caja de texto *Grados_F*, al valor equivalente en grados fahrenheit. Para hacer esto, simplemente llamamos al procedimiento que realiza la conversión de grados centígrados a grados fahrenheit. Esto es,

```
Grados_C_KeyPress (13)
```

Recuerde que este procedimiento se activa cada vez que pulsamos una tecla sobre la caja de texto y es la tecla pulsada el argumento que recibe como parámetro. La conversión se efectuaba cuando pulsábamos **Enter** (valor ASCII 13). Por ello, pasamos como argumento el valor 13.

Para escribir un procedimiento que ejecute estas operaciones, haga doble clic sobre el botón situado al lado de la caja de texto *Grados_C* para visualizar su ventana de código y escriba,

```
Sub Orden1_Click ()
  Grados_C.Text = Format$(Val(Grados_C.Text) - 1, "#,##0.00")
  Grados_C_KeyPress (13)
End Sub
```

Siguiendo el mismo razonamiento, el procedimiento para el otro botón es:

```
Sub Orden2_Click ()
  Grados_F.Text = Format$(Val(Grados_F.Text) - 1, "#,##0.00")
  Grados_F_KeyPress (13)
End Sub
```

Ejecute la aplicación y observe como funciona. Debido a que Windows está configurado para el idioma Español, en las cajas de texto aparece como separador de los decimales la coma. Esta característica queda reflejada por la constante **sDecimal** del fichero WIN.INI de Windows.

Esto hace que la función **Val** no actúe correctamente. Por ejemplo, si la caja de texto *Grados_C* contiene 10,5 la expresión Val(10,5) da un resultado de 10. La solución a este problema es pasar el argumento a la función **Val** en la notación inglesa (punto decimal).

REEMPLAZANDO EL SEPARADOR DECIMAL

El separador decimal, dependiendo del país, puede ser la coma o el punto. Por ejemplo en España se utiliza la coma y en EE. UU. se utiliza el punto. Esta diferencia causa un problema con la función **Val**, que asume el punto como separador decimal. La solución a este problema para los países que utilizan la coma como separador decimal es añadir la siguiente función como un procedimiento general.

```
' Función que convierte una cadena que representa un
' número con coma decimal, a un valor numérico.
Function Valor (ByVal Texto As String) As Double
  Dim I As Integer, L As Integer
  L = Len(Texto)
  If L = 0 Then Exit Function
  For I = 1 To L
    If Mid$(Texto, I, 1) = "," Then
      Mid$(Texto, I) = "."
    ElseIf Mid$(Texto, I, 1) = "." Then
      Mid$(Texto, I) = " "
    End If
  Next
  Valor = Val(Texto)
End Function
```


La función *Valor* devuelve como resultado el valor correspondiente a una cadena de caracteres que incluye la coma como separador decimal.

La función **Mid$**(*XS, n, m*) da como resultado de la cadena *XS* una subcadena de *m* caracteres de longitud, empezando por el *n-ésimo* carácter.

La sentencia **Mid$**(*XS, n*) = *YS* sustituye los caracteres que están a partir de la posición *n* en *XS* por los caracteres que hay en *YS*.

Observe que la función *Valor* recibe como argumento una cadena de caracteres que representa un valor numérico con coma de separador decimal y punto de separador de los miles, cambia la coma por el punto y sustituye los puntos de los miles por espacios que no son tenidos en cuenta por la función **Val**. El resultado es una cadena de caracteres que representa un valor con punto decimal válida para la función **Val** que realiza finalmente la conversión. El valor devuelto por la función *Valor* es el valor devuelto por la función Val, el cual es de tipo **Double**.

Para que esta función pueda ser accedida desde cualquier parte de la aplicación, la colocaremos en un módulo que denominaremos *Conver.bas*. Un procedimiento definido en un módulo puede llamarse desde cualquier parte de la aplicación.

Para crear la función *Valor* abra la ventana de código correspondiente a un nuevo módulo, orden **New Module** del menú **File**, y a continuación ejecute la orden **New Procedure** del menú **Code**; elija en la ventana de diálogo presentada la opción **Function** y escriba en la caja de texto de la misma el nombre *Conver*; finalmente pulse el botón **OK** y escriba el código correspondiente.

Para modificar un procedimiento general existente (**Sub** o **Function**), selecciónelo en la ventana de la aplicación, pulse **View Code**, se visualiza la ventana de código, seleccione "(general)" de la caja de objetos y a continuación seleccione el procedimiento de la caja de procedimientos.

EJECUTAR ÓRDENES DEL SISTEMA

Un botón puede utilizarse también para ejecutar una orden del sistema operativo o para ejecutar otro programa. Esto puede hacerse fácilmente utilizando la función **Shell**(*ordenS[, n]*). El argumento *ordenS* representa el nombre de la orden o del programa que se desea ejecutar y el argumento *n* es un valor entero (1, 2, 3, 4, 7) que hace que el proceso ejecutado se presente en una ventana normal, minimizada o maximizada, enfocada o sin enfocar. Por ejemplo,

```
Sub Libros_Click ()
  id = Shell("c: windows cardfile.exe libros.crd", 2)
End Sub
```

Cuando se pulse el botón correspondiente a este procedimiento, se ejecuta la aplicación *cardfile* de Microsoft Windows. Observe que la función retorna un valor *id* que identifica el programa arrancado e indica si se ejecutó satisfactoriamente. Este valor normalmente no es utilizado.

CAPÍTULO 5

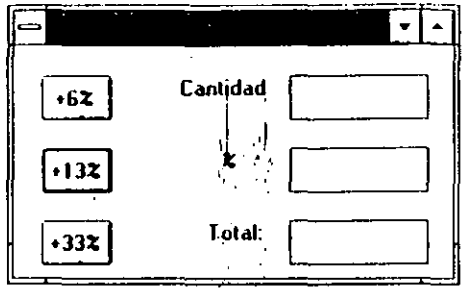
ARRAYS DE CONTROLES

INTRODUCCIÓN

Un *array de controles* es un conjunto de controles similares que comparten el mismo nombre y los mismos procedimientos. Los elementos de un array de controles son controles físicamente separados, cada uno con su propio conjunto de propiedades. Por ejemplo, supongamos una aplicación "Tanto por Ciento", formada por los controles especificados en la siguiente tabla:

	Caption	CtlName	Default	BackColor
Botón 1	+6%	TantoPorCiento	False	Por defecto
Botón 2	+13%	TantoPorCiento	True	Por defecto
Botón 3	+33%	TantoPorCiento	False	Por defecto
Etiqueta 1	Cantidad	Label1		Por defecto
Caja Texto 1		Text1		&H0000FFFF&
Etiqueta 2	%	Label2		Por defecto
Caja Texto 2		Text2		&H0000FFFF&
Etiqueta 3	Total:	Label3		Por defecto
Caja Texto 3		Text3		&H0000FFFF&

Esta tabla dará lugar, como se verá a continuación, al medio de comunicación que se presenta en la figura siguiente.

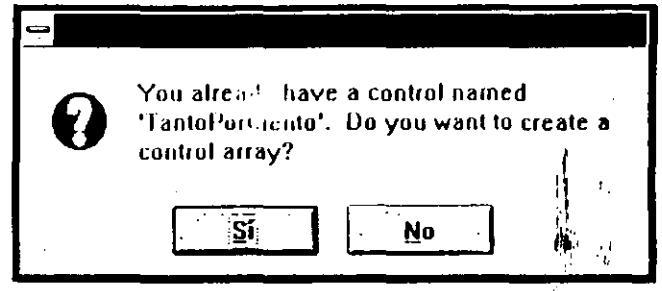


En este caso, se pretende calcular la cantidad *Total*, resultado de sumar a una *Cantidad* un determinado tanto por ciento (6, 13 o 33). No sería una buena solución, sino mas bien mala, escribir un procedimiento para cada botón (por ejemplo, *Botón6_Click*, *Botón13_Click* y *Botón33_Click*), porque los botones ejecutan esencialmente la misma acción, *Total = Cantidad + %Cantidad*, con una pequeña variación, el valor del tanto por ciento. La forma más fácil de tratar un caso como éste es asignar el mismo nombre a los botones, lo que da lugar a un array de controles, y escribir así un solo procedimiento.

Un array de controles se crea al asignar el mismo nombre a dos o mas controles durante el diseño de una aplicación, o bien, asignando un valor a la propiedad **Index** de un control. La última forma genera un array de un solo elemento, lo cual es útil cuando se quiere crear controles en tiempo de ejecución. Para crear un nuevo control en tiempo de ejecución, él debe ser un elemento de un array de control.

Cuando Visual Basic crea un array de controles, asigna por defecto los índices 0, 1, 2, ..., los cuales pueden modificarse a voluntad del usuario, cambiando el valor de la propiedad **Index**. Esta operación sólo puede realizarse durante el diseño.

Por ejemplo, dibuje el medio de comunicación anterior y asigne a los botones el nombre *TantoPorCiento*. Para ello, seleccione el botón *+6%* y asigne a su propiedad **CtlName** el valor *TantoPorCiento*. A continuación repita la misma operación para el botón titulado *+13%*. Cuando usted haga esto, Visual Basic presenta una caja como la de la figura diciendo: "Usted ya tiene un control nombrado 'TantoPorCiento'. ¿Quiere crear un array de controles?".



Responda sí y continúe con el resto de los botones. Si ahora hace doble clic sobre cada uno de los botones observará que el esquema del procedimiento asociado es el mismo, esto es, ahora, en lugar de tres procedimientos separados hay un sólo procedimiento como el siguiente:

```
Sub TantoPorCiento_Click (Index As Integer)

End Sub
```

Lo que ha hecho Visual Basic ha sido asignar a los botones el mismo nombre más un índice para diferenciarlos. Estos nombres para nuestro ejemplo son: *TantoPorCiento(0)*, *TantoPorCiento(1)* y *TantoPorCiento(2)*.

Cuando cualquier control del array reconoce un suceso, Visual Basic pasa el índice (**Index**) como un argumento extra. Este argumento siempre va antes que cualquier otro argumento.

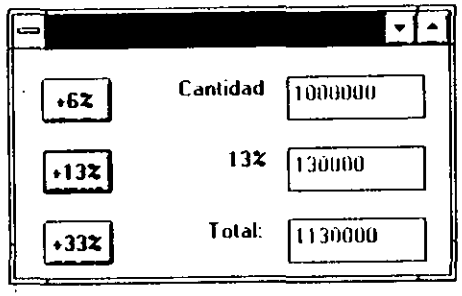
Con un array de controles, cada nuevo elemento hereda los procedimientos comunes al array.

A diferencia de un array ordinario, un array de controles

1. No se declara en el código.
2. Está limitado a una sola dimensión.
3. No tiene definido el límite superior (el límite inferior de un array de controles es cero).
4. No puede ser pasado como un único argumento (cada elemento debe ser pasado como un argumento separado).
5. Los índices no necesariamente tienen que ser consecutivos. Por ejemplo, *MiArray(1)* y *MiArray(7)* pueden ambos corresponder a controles, aunque no haya controles con los índices 2 a 6.

El valor de **Index** puede utilizarse dentro del procedimiento para determinar qué control ha recibido el suceso. Un elemento del array de controles puede referenciarse escribiendo *NombreArray(Index)*, nomenclatura que puede utilizarse en cualquier lugar donde pueda especificarse el nombre del control.

Continuando con el ejemplo, nuestro propósito es escribir una cantidad y al pulsar **Enter** deseamos que automáticamente se hagan los cálculos para el 13%. La cantidad tecleada aparecerá en la caja de texto primera, el cálculo del tanto por ciento aparecerá en la segunda caja, la etiqueta de esta caja variará de acuerdo con el tanto por ciento (6%, 13% o 33%) y la suma de las dos cantidades anteriores aparecerá en la tercera caja.



Para conseguir que al pulsar **Enter** automáticamente se hagan los cálculos para el 13%, hay que hacer que el botón *+13%* sea el botón por defecto, lo que implica poner la propiedad **Default** del mismo a valor **True**. El botón por defecto se distingue de los demás, porque aparece rodeado con un borde más negro.

Según lo expuesto, el cuerpo para el procedimiento *TantoPorCiento* conducido por el suceso **Click**, es el siguiente:

```
Sub TantoPorCiento_Click (Index As Integer)
  Dim CantIni As Double, T As Double, Tp As Double
  Tp = Val(TantoPorCiento(Index).Caption)
  CantIni = Val(Text1.Text)
  T = Tp / 100 * CantIni
  Label2.Caption = Str$(Tp) + "%"
  Text2.Text = Format$(T)
  Text3.Text = Format$(CantIni + T)
End Sub
```

Observe como el tanto por ciento *Tp* a aplicar, se obtiene convirtiendo el título del botón a valor numérico.

Cuando uno de los controles de un array reconoce un suceso, Visual Basic llama al procedimiento común y le pasa un argumento adicional (propiedad **Index**) que identifica el control que actualmente ha reconocido el suceso. Por ejemplo, si hacemos clic en el botón *+33%* Visual Basic pasa el valor 2 para **Index**. Por lo tanto, *Val(TantoPorCiento(Index).Caption)* da como resultado 6 para **Index** igual a 0, 13 para **Index** igual a 1 y 33 para **Index** igual a 2.

La sentencia *Label2.Caption = Str$(Tp) + "%"* modifica la etiqueta de la caja *Text2* al valor 6%, 13% o 33% dependiendo de *Tp*.

Creando controles en tiempo de ejecución

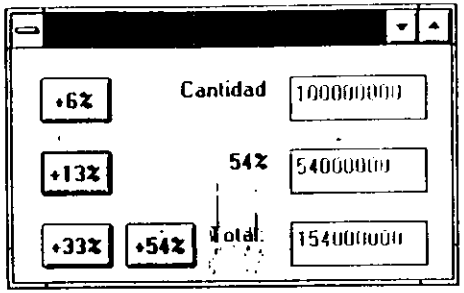
Un array de controles tiene al menos un elemento y puede llegar a tener hasta 255. Cada elemento tiene su propio conjunto de propiedades, aunque comparten el mismo código. Puesto que el código a compartir tiene que existir, no es posible crear durante la ejecución de la aplicación un nuevo control, si no existe un array de controles.

Para añadir un nuevo control a un array de controles durante la ejecución de una aplicación, utilizaremos la sentencia **Load** y para eliminarlo utilizaremos la sentencia **Unload**. No se puede utilizar **Unload** para eliminar un control que no haya sido creado en tiempo de ejecución. Por ejemplo,

```
Load TantoPorCiento(3)   'Añade un control con índice 3
Unload TantoPorCiento(3) 'Elimina un control de índice 3
```

Cuando se añade un nuevo elemento a un array de controles, todas las propiedades excepto **Visible**, **Index**, y **TabIndex** son copiadas del elemento más bajo en el array. Esto hace que cuando añadamos varios controles, todos sean del mismo tamaño y se apilen en el mismo lugar, ya que las propiedades **Left** (distancia desde la izquierda), **Top** (distancia desde arriba), **Width** (anchura) y **Height** (altura) son idénticas. Por otra parte, ninguno de los controles creados serán visibles ya que se crean con la propiedad **Visible** a valor **False**.

Como ejemplo, vamos a considerar el siguiente caso sobre la aplicación anterior: cuando escribamos una cantidad igual o superior a 100.000.000, aparecerá un nuevo botón titulado *+54%*, como se indica en la figura siguiente, y cuando escribamos una cantidad menor de 100.000.000 este botón, si está en la forma, desaparecerá. Contemplar los casos de no añadir el botón cuando ya esté añadido y no eliminarlo cuando ya esté eliminado.



El procedimiento común será ahora el siguiente:

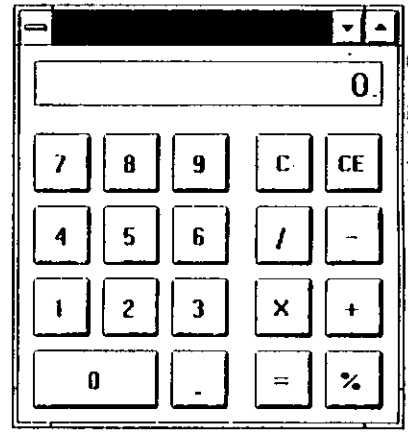
```
Sub TantoPorCiento_Click (Index As Integer)
  Dim CantIni As Double, T As Double, Tp As Double
  Static Botón54 As Integer
  Tp = Val(TantoPorCiento(Index).Caption)
  CantIni = Val(Text1.Text)
  T = Tp / 100 * CantIni
  Label2.Caption = Str$(Tp) + "%"
  Text2.Text = Format$(T)
  Text3.Text = Format$(CantIni + T)
  If CantIni >= 100000000 And Botón54 = 0 Then
    Load TantoPorCiento(3)
    TantoPorCiento(3).Left = 840
    TantoPorCiento(3).Top = 1440
    TantoPorCiento(3).Caption = "-54%"
    TantoPorCiento(3).Visible = -1
    Botón54 = 1
  ElseIf CantIni < 100000000 And Botón54 = 1 Then
    Unload TantoPorCiento(3)
    Botón54 = 0
  End If
End Sub
```

La variable estática *Botón54* está a cero cuando el botón *+54%* no está en la forma y se pone a 1 cuando este botón está sobre la forma. Esto permitirá no añadir el botón cuando ya esté añadido y no eliminarlo cuando ya esté eliminado.

Observe que las propiedades como **Width** (ancho) y **Height** (alto) no se han especificado puesto que son heredadas. En cambio, sí se han especificado las propiedades **Left** y **Top** que indican la posición del control dentro de la forma, la propiedad **Caption** que especifica el título del botón, y la propiedad **Visible** que hace que el control creado sea visible.

DISEÑO DE UNA CALCULADORA

A continuación vamos a realizar el diseño de una calculadora como la de la figura siguiente.



Antes de crear una aplicación debemos responder algunas preguntas como las siguientes:

- ¿Qué objetos forman el medio de comunicación?
- ¿Qué sucesos hacen que el medio de comunicación responda?
- ¿Cuáles son los pasos a seguir para un desarrollo ordenado?

Objetos

- La calculadora incluye los siguientes objetos:
- Una forma que permita implementar nuestro medio de comunicación.
- Una etiqueta (pantalla de la calculadora).
- Diecinueve botones de órdenes distribuidos de la forma siguiente:
  - dígitos del 0 al 9
  - punto decimal
  - operaciones +, −, ×, /, =, y %
  - operaciones de borrar todo y borrar última entrada (C y CE).

Sucesos

Haciendo clic sobre las teclas de los dígitos visualizaremos un número en la pantalla. Un número puede contener un punto decimal. La entrada de un número finalizará al hacer clic sobre una de las teclas de operación. Por lo tanto, el suceso al que tiene que responder cada uno de los botones es un clic del ratón (o su equivalente, seleccionar el botón con **Tab** y pulsar la barra espaciadora), que en Visual Basic se denomina suceso **Click**.

Pasos a seguir durante el desarrollo

1. Crear una nueva aplicación.
2. Mover y ajustar el tamaño por defecto de la forma.
3. Dibujar los controles.
4. Definir las propiedades de la forma y de los controles.
5. Escribir el código para cada uno de los objetos.
6. Guardar la aplicación.
7. Verificar la aplicación.
8. Crear un fichero ejecutable.

Diseño de la forma y de los controles

Conocidos los objetos y los sucesos, procedemos a dibujar el medio de comunicación entre el usuario y la aplicación. Para ello, abrimos una nueva aplicación **(New Project)**, si aún no lo hemos hecho, ajustamos la forma y dibujamos los controles.

La propiedad **Caption** nos permite cambiar el título a la forma y a los controles. De esta manera podremos poner a la forma el título "Calculadora" y a cada una de las teclas (controles) el carácter correspondiente a su función.

La pantalla es una etiqueta titulada "0.", ajustada a la derecha y con un borde. Las propiedades que hay que especificar para conseguir lo expuesto son:

Alignment	1 (justificado a la derecha)
BackColor	&H0000FFFF&
BorderStyle	1 (una línea fija)
Caption	0.
CtlName	Pantalla
FontSize	13,5

Para referirnos en el código a cada uno de los controles hay que asignarles un nombre. La propiedad **CtlName** de cada control permite realizar esta operación. Pero antes, consideremos si hay controles que tienen un comportamiento similar y por lo tanto pueden compartir los mismos procedimientos.

En el caso de la calculadora, hay 10 teclas cuya finalidad es visualizar un dígito en la pantalla, lo que deja claro que su función es la misma, lo que quiere decir que pueden compartir los mismos procedimientos. Igualmente sucede con el grupo de teclas +, −, ×, /, =. El resto de las teclas actúan independientemente.

Para que varios controles compartan los mismos procedimientos, deben formar parte de un array de controles, lo que implica que tengan el mismo nombre.

Según lo expuesto, las propiedades para las teclas son las siguientes:

	Caption	**CtlName**	**FontSize**
Teclas 0 a 9	0, 1, ...,9	Dígito (array de controles)	9,75
Teclas +, −, ×, /, =	+, −, ×, /, =	Operación (array de controles)	12
Punto decimal	.	PuntoD	13,5
Tanto por ciento	%	TantoPorCiento	9,75
Puesta a cero total	C	Inicializar	9,75
Puesta a cero última entrada	CE	BorrarEntrada	9,75

Una vez que se ha puesto nombre a todos los controles y se han especificado el resto de las propiedades de interés, el paso siguiente es unir el código correspondiente a cada uno de los controles para que cada uno de ellos cumpla su función.

Escribir el código

¿Por donde se empieza?. En una programación conducida por sucesos se puede empezar por cualquier control.

Por ejemplo, empiece con las teclas correspondientes a los dígitos 0 a 9, y pregúntese: ¿qué tiene que suceder cuando se haga clic sobre un dígito?. Cuando esto suceda se espera que el dígito aparezca en la pantalla de la calculadora. Esto se consigue escribiendo un procedimiento que responda al suceso **Click** y asigne el título de la tecla (0,1,...) al título de la pantalla. Para crear un procedimiento como éste hay que realizar los pasos siguientes:

1. Abrir la ventana de código. Para ello, haga un doble clic sobre cualquier control del array *Dígito*, o seleccione el control (enfóquelo) y pulse **F7**. Visual Basic visualiza el siguiente esquema:

```
Sub Dígito_Click (Index As Integer)

End Sub
```

   Observe que el nombre del objeto es *Dígito(Index)* y que el suceso es **Click**.

2. Escribir el código. Anteriormente se ha indicado que hay que asignar el título de la tecla pulsada (0,1,...) al título de la pantalla. Para ello, escriba como cuerpo del procedimiento la sentencia,

```
Pantalla.Caption = Dígito(Index).Caption
```

Este procedimiento será compartido por todas las teclas agrupadas en el array de controles *Dígito*. Cuando un control del array reconoce el suceso **Click**, Visual Basic pasa el índice (**Index**) como un argumento extra para identificar la tecla que ha sido pulsada.

El procedimiento tal cual se ha escrito, asigna un sólo dígito a la pantalla de la calculadora. Para asignar un dígito tras otro hay que modificar el código de forma que cada nuevo dígito tecleado sea añadido a los ya existentes en la pantalla. Esto se consigue modificando la sentencia anterior así:

```
Sub Dígito_Click (Index As Integer)
  Dim Número As String
  Número = Pantalla.Caption
  Pantalla.Caption = Número + Dígito(Index).Caption
End Sub
```


Guarde la aplicación y ejecútela. Observará que ahora todos los números tecleados empiezan por "0.". Esto indica que cada vez que el usuario vaya a introducir un nuevo número, la pantalla debe ser inicializada. Este caso se da al principio, y después de teclear un operador. Para detectarlo introduciremos una nueva variable denominada *UltimaEntrada* que tendrá el valor "DÍGITO" si lo último tecleado ha sido un dígito, y otro valor en otro caso.

```
If UltimaEntrada <> "DÍGITO" Then
  Pantalla.Caption = ""
End If
```

Otro detalle a tener en cuenta es que los ceros iniciales no son significativos, por lo que no se tendrán en cuenta. Esto es,

```
If Pantalla.Caption = "0." And Dígito(Index).Caption = "0" Then
  Exit Sub
End If
```

Las variables, como *UltimaEntrada*, que tengan que ser compartidas por todos los procedimientos de la forma tienen que ser declaradas a nivel de la forma. Para declarar una variable a este nivel, abra la ventana de código para un objeto y seleccione "(general)" de la caja de objetos y "(declarations)" de la caja de sucesos para ese objeto.

```
Dim UltimaEntrada As String
```

Con las modificaciones introducidas, el código queda como sigue:

```
Sub Dígito_Click (Index As Integer)
  Dim Número As String
  If Pantalla.Caption = "0." And Dígito(Index).Caption = "0" Then
    Exit Sub
  End If
  If UltimaEntrada <> "DÍGITO" Then
    Pantalla.Caption = ""
  End If
  Número = Pantalla.Caption
  Pantalla.Caption = Número + Dígito(Index).Caption
  UltimaEntrada = "DÍGITO"
End Sub
```

Para añadir decimales a un número se pulsa la tecla del punto decimal. Antes de escribir el código para esta tecla analicemos los casos que se pueden dar:

1. El usuario introduce un número que no tiene parte entera y pulsa primer tecla el punto.

2. El número tiene parte entera.

3. No aceptar un segundo punto decimal. Para detectar este caso introduciremos una nueva variable denominada *PuntoDecimal* que tendrá el valor SI, si el número ya tiene un punto decimal o un valor NO si no lo tiene. Defina esta variable y estas constantes a nivel de la forma.

```
Dim PuntoDecimal As Integer
Const SI = -1
Const NO = 0
```

De acuerdo con lo expuesto el procedimiento para el punto decimal es el siguiente:

```
Sub PuntoD_Click ()
  If UltimaEntrada <> "DÍGITO" Then
    Pantalla.Caption = "0."
  Elseif PuntoDecimal = NO Then
    Pantalla.Caption = Pantalla.Caption + "."
  End If
  PuntoDecimal = SI
  UltimaEntrada = "DÍGITO"
End Sub
```

La introducción del punto decimal obliga a realizar algunos ajustes en el procedimiento *Dígito_Click* para poder introducir números de la forma 0.0... y para permitir el punto decimal para un nuevo número.

```
Sub Dígito_Click (Index As Integer)
  Dim Número As String
  If Pantalla.Caption = "0." And Dígito(Index).Caption
      = "0" And PuntoDecimal = NO Then
    Exit Sub
  End If
  If UltimaEntrada <> "DÍGITO" Then
    Pantalla.Caption = ""
    PuntoDecimal = NO
  End If
  Número = Pantalla.Caption
  Pantalla.Caption = Número + Dígito(Index).Caption
  UltimaEntrada = "DÍGITO"
End Sub
```

(Nota: si la primera línea *If ... Then* aparece partida en dos, es debido a que no cabe en una sola línea. En Visual Basic se escribirá en una sola línea. Tenga en cuenta ésto para casos similares a lo largo de esta obra.)

Una vez que el usuario ha tecleado un número, lo lógico es que pulse a continuación la tecla correspondiente a la operación a realizar. Esto es, el usuario hará clic sobre una de las teclas +, −, ×, / o =, agrupadas en el array *Operación*. En este instante la calculadora tiene que recordar el operador introducido.

```
Operador = Operación(Index).Caption
```

A continuación el usuario introducirá otro número. Para presentar en pantalla el nuevo número, ésta tiene que ser inicializada. Para enterarnos de este detalle, después de introducir un operador haremos que la variable *UltimaEntrada* tome el valor "OPERADOR".

Por otra parte, si la calculadora además del operador no recuerda el primer operando, este se perderá cuando se inicialice la pantalla para introducir el segundo operando.

```
Operando1 = Val(Pantalla.Caption)
```

Recopilando lo hasta ahora expuesto y pensando un poco mas allá, el proceso total se puede resumir en los siguientes puntos:

1. Primero, un usuario introduce un número (operando).
2. A continuación, introduce un operador. La calculadora recuerda el número y el operador.
3. Después introduce un segundo número.
4. Y cuando introduce otro operador, la calculadora:
   - Recuerda el segundo número.
   - Visualiza en la pantalla el resultado producido por el operador sobre los dos operandos.
   - Recuerda el resultado, como si se hubiera introducido un operando, y el último operador.
5. Si se introduce un operador a continuación de otro se recuerda el último.
6. Con respecto a una operación cualquiera la calculadora debe saber cuando se introduce el operando primero y cuando se introduce el segundo, para almacenarlos en lugares diferentes y para saber si ya tiene que realizar la operación. Para esto utilizaremos una variable *NumOperandos* que valdrá 1 cuando se haya introducido el operando primero y 2 cuando se haya introducido el segundo.

7. El resultado de una operación queda disponible como operando para una siguiente siguiente operación.

Para crear este procedimiento, haga un doble clic en una de las teclas +, -, ×, / o =, o bien, enfóquela y pulse **F7**. Esto hace que se visualice la ventana de código correspondiente, con el esquema que se indica a continuación.

```
Sub Operación_Click (Index As Integer)

End Sub
```

Observe que el objeto sobre el que ha hecho doble clic, pertenece al array de controles *Operador* y que el suceso es **Click**.

Seleccione "(general)" de la caja de objetos y "(declarations)" de la caja de sucesos para ese objeto, y declare las variables,

```
Dim NumOperandos As Integer
Dim Operando1 As Double, Operando2 As Double
Dim Operador As String * 1
```

Escriba el código siguiente, resultado del análisis que se acaba de hacer.

```
Sub Operación_Click (Index As Integer)
  If UltimaEntrada = "DÍGITO" Then
    NumOperandos = NumOperandos + 1
  End If
  If NumOperandos = 1 Then
    Operando1 = Val(Pantalla.Caption)
  ElseIf NumOperandos = 2 Then
    Operando2 = Val(Pantalla.Caption)
    Select Case Operador
      Case "+": Operando1 = Operando1 + Operando2
      Case "-": Operando1 = Operando1 - Operando2
      Case "X": Operando1 = Operando1 * Operando2
      Case "/": Operando1 = Operando1 / Operando2
      Case "=": Operando1 = Operando2
    End Select
    Pantalla.Caption = Str$(Operando1)
    NumOperandos = 1
  End If
  UltimaEntrada = "OPERADOR"
  Operador = Operación(Index).Caption
End Sub
```


La siguiente tecla que vamos a programar es la del tanto por ciento. Para ello, piense primero como trabaja esta tecla. Supongamos que ha llegado a la conclusión de que la forma general de actuar es:

Operando_1 Operador Operando_2 %

Por ejemplo, si se teclea *1000 + 5 %* se visualiza *1050*.

En el caso de pulsar repetidas veces la tecla %, la calculadora sólo responderá a la primera.

En función de lo expuesto, el procedimiento asociado con esta tecla es el siguiente:

```
Sub TantoPorCiento_Click ()
  Dim Resultado As Double
  If UltimaEntrada = "DÍGITO" Then
    Resultado = Operando1 * Val(Pantalla.Caption) / 100
    Pantalla.Caption = Str$(Resultado)
    Operación_Click (0) 'para "=", Index es 0
    UltimaEntrada = "OPERADOR"
  End If
End Sub
```

La sentencia *Resultado = Operando1 * Val(Pantalla.Caption) / 100* calcula el tanto por ciento especificado por *Pantalla.Caption* de la cantidad especificada por *Operando1*. Por ejemplo, si se teclea *1000 + 5 %*, *Resultado = 50*.

El resultado se lleva a la pantalla como si el segundo operando de la operación especificada (en el ejemplo +) se hubiera introducido. Para que se realice automáticamente esta operación, de acuerdo con el ejemplo *1000 + 50*, se llama al procedimiento *Operación_Click* como si la tecla "=" se hubiera pulsado.

Cuando se pulse la tecla C es por que se quiere inicializar la calculadora, esto es, como si la hubiéramos acabado de encender. Entonces, lo que tiene que suceder es que en la pantalla aparezca "0." y que las variables definidas a nivel de la forma sean inicializadas, para lo que se invocará al procedimiento *Form_Load*.

```
Sub Inicializar_Click ()
  Pantalla.Caption = "0."
  Form_Load
End Sub
```

Cuando se inicia la aplicación se ejecuta el procedimiento *Form_Load* asociado a la forma. Por lo tanto, este procedimiento sirve para inicializar las varia-

bles necesarias cuando se pone en marcha la calculadora y cuando se pulsa la tecla C.

```
Sub Form_Load ()
  NumOperandos = 0
  Operando1 = Operando2 = 0
  UltimaEntrada = "NINGUNA"
  Operador = ""
  PuntoDecimal = NO
End Sub
```

Cuando se pulse la tecla CE lo que se quiere es borrar el último dato tecleado (dato actual de la pantalla), porque nos hemos equivocado y tenemos que repetirlo. En este caso, lo único que hay que hacer es inicializar la pantalla a cero y poner las variables afectadas al valor correspondiente en ese instante.

```
Sub BorrarEntrada_Click ()
  Pantalla.Caption = "0."
  PuntoDecimal = NO
  UltimaEntrada = "CE"
End Sub
```

Para terminar, vamos a fijarnos en un último detalle. Cuando inicializamos la calculadora no es posible realizar una operación como, por ejemplo, -3×5 (no piense en operaciones como 5×-3, ya que, como es el último operador el que se tiene en cuenta, lo que se realiza es $5 - 3$). Este caso se da cuando sin haber tecleado operando alguno, tecleamos un signo – con la finalidad de que el primer operando sea negativo. Para dar solución a este caso particular añada el siguiente código al procedimiento *Operación_Click*.

```
Sub Operación_Click (Index As Integer)
  If NumOperandos = 0 And Operación(Index).Caption = "-" Then
    UltimaEntrada = "DÍGITO"
  End If
  If UltimaEntrada = "DÍGITO" Then
  ...

End Sub
```

Observe que para visualizar el resultado se convierte el valor numérico resultante a cadena de caracteres utilizando la función **Str$**. Si quiere representar el resultado bajo un determinado formato, utilice la función **Format$** en lugar de la función **Str$**.

Es importante recordar que si en lugar de utilizar **Str$** se utiliza la función **Format$**, la cadena resultante utiliza la notación correspondiente a nuestro idioma, esto es, coma decimal.

Según se vio en el capítulo anterior, cuando se convierte una cadena a un valor numérico utilizando la función *Val*, la conversión finaliza cuando se localiza una coma, lo que origina la pérdida de la parte decimal. Esto es debido a que esta función toma su argumento como si estuviera en notación inglesa (punto decimal). Este problema se solucionó entonces, implementando una función *Valor* que convierte una cadena de caracteres a un valor numérico, y que interpreta el punto, como separador de los miles y la coma, como separador de los decimales.

Según lo expuesto anteriormente, si desea que su calculadora utilice la coma como separador de los decimales, sustituya la función **Str$** por la función **Format$** para convertir un valor numérico a una cadena de caracteres y la función **Val** por la función *Valor* para convertir una cadena de caracteres a un valor numérico (véa el capítulo anterior).

Una vez que haya ejecutado y comprendido esta aplicación, como ejercicio intente añadir algunas funciones más a la calculadora. Por ejemplo,

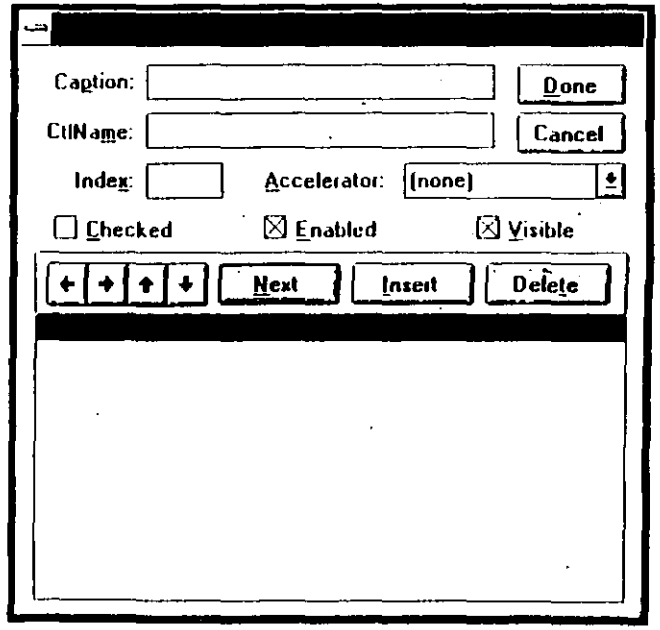
1. Añadir las teclas de memoria,
   - sumar al contenido de la memoria (M+)
   - restar del contenido de la memoria (M−)
   - leer el contenido de la memoria (MR).
2. Añadir las funciones matemáticas más usuales.
3. Permitir utilizar, además del ratón, el teclado numérico para introducir datos.

CAPÍTULO 6

TÉCNICAS DE DISEÑO

DISEÑO DE UN MENÚ

Igual que una caja de texto o un botón, un menú es un control. Para diseñar un menú utilizaremos la ventana de diseño de menús que se puede ver en la figura siguiente.

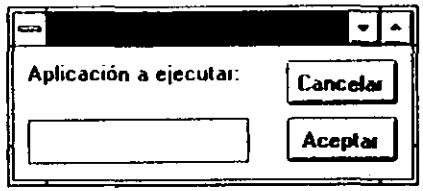


Para crear un menú los pasos a ejecutar son los siguientes:

ejecutar. Este nombre corresponderá a un fichero existente en el disco con extensión .COM, .EXE, .BAT, o .PIF; por ejemplo, WINMINE que arranca el juego "busca minas" que va con Windows. Por lo tanto, el procedimiento asociado con la orden *Ejecutar* es el siguiente:

```
Sub FicheroEjecutar_Click ()
  Ejecutar.Show
End Sub
```

Cree una nueva forma y asígnela el título y el nombre *Ejecutar*. Añada a la forma una etiqueta "Aplicación a ejecutar:", una caja de texto en la que el usuario pueda escribir el nombre de la aplicación a ejecutar, y dos botones con los nombres *Cancelar* y *Aceptar*.



Una vez que el usuario escriba el nombre de la aplicación a ejecutar, pulsará el botón *Aceptar*; en este instante la aplicación arranca y se ejecuta normalmente.

```
Sub Aceptar_Click ()
  Dim Id As Integer
  Id = Shell(Text1.Text, 1)
  Text1.SetFocus
  Text1.Text = ""
  Ejecutar.Hide
End Sub
```

Este procedimiento, primero ejecuta la orden **Shell** que arranca la aplicación referenciada por *Text1.Text* y después devuelve el foco, que está en el botón *Cancelar*, a la caja de texto, elimina el texto existente en la misma, y oculta la forma.

Si el usuario pulsa el botón *Cancelar*, no se ejecuta nada y se oculta la forma.

```
Sub Cancelar_Click ()
  Text1.SetFocus
  Text1.Text = ""
  Ejecutar.Hide
End Sub
```

CAPÍTULO 8

FICHEROS DE DATOS

INTRODUCCIÓN

Muchas aplicaciones requieren almacenar y recuperar datos de una sesión a otra. Los ficheros de datos permiten almacenar información que el usuario introduce a través de una caja de diálogo y por lo tanto, son también la fuente de información para visualizar datos sobre una forma o para escribir datos en una impresora.

Por ejemplo, recuerde que en Capítulo 6 desarrollamos un editor cuyo menú *Fichero* no soportaba las órdenes *Abrir* y *Guardar*. Como consecuencia, al finalizar un trabajo de edición, no era posible guardar la información editada y por lo tanto, tampoco podríamos recuperar dicha información en una nueva sesión. El mismo problema se nos presentó cuando diseñamos la base de datos para llevar la contabilidad de los libros de nuestra biblioteca en el Capítulo 7. Estos problemas serán solucionados en este capítulo.

Un fichero de datos se almacena como un fichero separado en el disco, esto es, un fichero independiente de los ficheros que almacenan la aplicación.

Los tipos de ficheros de Visual Basic así como las sentencias para la manipulación de los mismos coinciden con los tipos y sentencias soportados por QBASIC o QuickBASIC. No obstante, existen muchas diferencias, en la forma de comunicarse con el usuario.

TIPOS DE FICHEROS EN VISUAL BASIC

Visual Basic permite crear tres clases diferentes de ficheros de datos: secuenciales, aleatorios y binarios.

El tipo más simple de fichero de datos es el fichero secuencial. Este es un fichero de texto que puede almacenar registros de cualquier longitud. Cuando la información se escribe registro a registro, estos son colocados uno a continuación de otro y cuando se lee, se empieza por el primer registro y se continua al siguiente hasta alcanzar el final. Generalmente, se utilizan como ficheros de texto en los que se escribe toda la información desde el principio hasta el final y se lee de la misma forma. Veremos esto prácticamente cuando modifiquemos nuestro editor para que permita guardar la información editada. Las sentencias y funciones más comunes para tratar este tipo de ficheros son:

Open, Print #, Write #, Close, Input #, Line Input #, Input$, Eof

Un fichero aleatorio consiste en un conjunto de registros de la misma longitud, los cuales pueden ser accedidos en cualquier secuencia. Cada registro individual se identifica con un único número y puede ser leído, escrito o actualizado. Un fichero aleatorio es conocido también como un fichero de acceso al azar, de acceso directo o fichero relativo. La utilización de ficheros aleatorios la veremos prácticamente cuando modifiquemos la aplicación referente a la base de datos de libros. Las sentencias y funciones más comunes para tratar este tipo de ficheros son:

Open, Type...End Type, Put, Get, Close, LOF, Loc

Los ficheros binarios ofrecen la posibilidad de tratar cualquier fichero como una secuencia numerada de caracteres, independientemente de la estructura del mismo. Los caracteres ocupan las posiciones 1, 2, 3, etc. Por ejemplo, si quisiéramos copiar un fichero ejecutable (.EXE) en otro, tendremos que tratarlo como un fichero binario para leerlo carácter a carácter (byte a byte). Las sentencias y funciones más comunes para tratar este tipo de ficheros son:

Open, Put, Get, Close, Seek, Input$

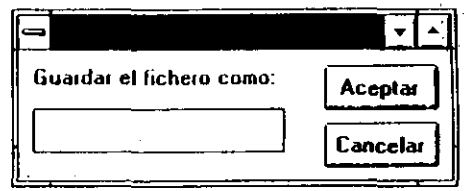
UTILIZACIÓN DE FICHEROS SECUENCIALES

Vamos a añadir soporte a nuestra aplicación *Editor*, para lo que añadiremos al menú *Fichero* las órdenes *Abrir* y *Guardar*.

Cargue la aplicación *Editor* que realizó en el Capítulo 6 y añada al menú *Fichero* las órdenes con las propiedades que se indican en la tabla siguiente. Recuerde que tres puntos a continuación de una orden, es una forma de indicar al usuario que cuando ejecute la orden, se va a visualizar una caja de diálogo.

Caption	CtlName	Accelerator
Abrir...	FicheroAbrir	Ctrl+A
Guardar...	FicheroGuardar	Ctrl+G

Cuando el usuario seleccione la orden *Guardar*, queremos que se visualice una caja de diálogo como la que se presenta a continuación.



Cree una nueva forma, asígnela el título *Guardar Fichero* y el nombre *GuardarF*. A continuación añada los siguientes controles:

	Caption	CtlName	Text
Etiqueta	Guardar el fichero como:	Label1	
Caja de texto		NombreFg	(nada)
Botón 1	Aceptar	Aceptar	
Botón 2	Cancelar	Cancelar	

Como se trata de una caja de diálogo podemos eliminar los botones de minimizar y maximizar la forma. Para ello, ponga las propiedades **MinButton** y **MaxButton** de la forma a valor **False**. También, y con el fin de que el usuario no pueda variar el tamaño de la forma, ponga la propiedad **BorderStyle** a valor 3.

Para que se visualice la forma que acabamos de crear cuando el usuario haga clic en la orden *Guardar*, hay que unir a esta orden el siguiente procedimiento:

```
Sub FicheroGuardar_Click ()
  GuardarF.Show
End Sub
```

A continuación, unimos el procedimiento correspondiente al botón *Cancelar*. Cuando el usuario pulse este botón, nosotros queremos limpiar la caja de texto, enfocarla, para que la siguiente vez que se visualice la forma esté enfocada la caja de texto y no el botón *Cancelar*, ocultar la forma y enfocar la ventana del editor. Para ello, abra la ventana de código para el botón *Cancelar* y escriba:

```
Sub Cancelar_Click ()
  NombreFg.Text = ""
  NombreFg.SetFocus
  GuardarF.Hide
  Form1.Text1.SetFocus
End Sub
```

Una vez que el usuario haya finalizado la edición, decidirá guardar la información escrita en un fichero en disco. Para ello, ejecutará la orden *Guardar* del menú *Fichero* y escribirá en la caja de texto el nombre de fichero con el que quiere guardar la información; después pulsará la orden *Aceptar*. En nuestro caso, la información de la caja de texto viene dada por *Form1.Text1.Text* y el nombre del fichero viene dado por *GuardarF.NombreFg.Text*.

Escribir en un fichero

Para realizar el proceso de guardar la información en el fichero, hay que realizar tres pasos:

1. *Abrir el fichero para escribir* (**Output**) utilizando la sentencia **Open**. En este caso crearemos un fichero secuencial. La sintaxis de esta sentencia es:

   Open *nombre-fich* [**For** *modo*] **As** [#]*nº fichero*

   nombre-fich es el nombre del fichero (puede incluir el camino);
   modo puede ser Output, Input, Append, Binary, o Random;
   nº fichero es un número entre 1 y 255 para referenciar el fichero.

2. *Escribir los datos en el fichero* utilizando la sentencia **Write #** o **Print #**. En nuestro caso utilizaremos **Print** cuya sintaxis es:

   Print #*nº fichero, expresiones*

   nº fichero es el número con el que se abrió el fichero para escribir;
   expresiones es una lista de expresiones numéricas y/o de caracteres separadas por espacios, comas o puntos y comas.

3. *Cerrar el fichero* mediante la sentencia **Close**. La sintaxis de esta sentencia es:

Close [[#]*nº fichero* [,[#]*nº fichero*]...]

nº fichero es el número con el que se abrió el fichero.

Continuemos con la aplicación. Abra la ventana de código correspondiente al botón *Aceptar* y escriba el procedimiento siguiente:

```
Sub Aceptar_Click ()
  'Escribir en el fichero
  Open NombreFg.Text For Output As #1
  Print #1, Form1.Text1.Text
  Close #1
  'Ocultar la forma
  NombreFg.Text = ""
  NombreFg.SetFocus
  GuardarF.Hide
End Sub
```

Este procedimiento tiene dos partes: escribir en el fichero y ocultar la forma. La sentencia **Open** abre el fichero referenciado por *NombreFg.Text* para escribir (**Output**) y le asigna el número 1. Si el fichero no existe se crea y si existe se destruye y se crea de nuevo. La sentencia **Print #1** escribe en el fichero número 1, la información referenciada por *Form1.Text1.Text* la cual se corresponde con la información editada; el fichero está creado. Finalmente la sentencia **Close #1** cierra el fichero número 1. Un fichero cerrado no puede ser accedido; para acceder al mismo, hay que volverlo a abrir con igual o diferente número.

Si al abrir un fichero existente para escribir no queremos que su contenido se destruya, tenemos que abrirlo en modo **Append** en lugar de modo **Output**. Esta modalidad permite añadir la nueva información al final del fichero.

Control de errores

Cuando se abre un fichero de esta forma, cabe la posibilidad de que se produzca un error porque el usuario ha especificado un camino incorrecto o porque el nombre no es correcto sintácticamente. Para manipular tales errores y cualquier otro tipo de error, vamos a introducir la sentencia.

On Error GoTo *etiqueta*

donde *etiqueta* identifica a la primera línea de la rutina que manipula el error.

Una vez que se ejecute la sentencia **On Error GoTo**, cualquier error que suceda en el procedimiento en ejecución, invocará a la rutina de error, donde nosotros manipularemos el mismo a nuestra conveniencia.

La sentencia **Resume** permite continuar la ejecución del programa después de realizar un procedimiento de recuperación de un error. Puede utilizarse de cualquiera de las formas siguientes:

Resume [0]	La ejecución se reanuda en la sentencia que provocó el error.
Resume Next	La ejecución se reanuda en la sentencia inmediatamente posterior a la que provocó el error.
Resume *etiqueta*	La ejecución se reanuda a partir de la etiqueta o número de línea especificado.

Haciendo uso de estas sentencias modifique el procedimiento *Aceptar_Click* de la forma siguiente:

```
Sub Aceptar_Click ()
  On Error GoTo RutinaDeError
  'Escribir en el fichero
  Open NombreFg.Text For Output As #1
  Print #1, Form1.Text1.Text
  Close #1
  'Ocultar la forma
  NombreFg.Text = ""
  NombreFg.SetFocus
  GuardarF.Hide
  Form1.Text1.SetFocus
Salir:
  Exit Sub
RutinaDeError:
  MsgBox "Error al abrir el fichero", 48, "Editor"
  NombreFg.SetFocus
  Resume Salir
End Sub
```

Observe que hemos colocado la sentencia **On Error** al principio del procedimiento para que cuando se abra el fichero, si el camino no existe o el nombre no es correcto, se invoque a la rutina de manipulación del error referenciada por la etiqueta *RutinaDeError*.

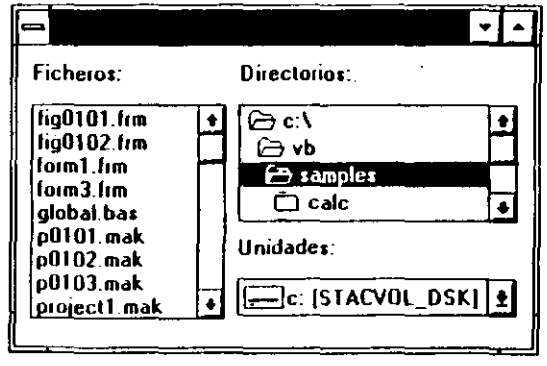
La rutina de error visualiza en una forma titulada "Editor", el mensaje "Error al el fichero" y enfoca de nuevo la caja de texto *NombreFg* para permitir al usu. orregir el error.

CONTROLES RELATIVOS A FICHEROS

El primer paso para leer el contenido de un fichero es obtener el nombre del mismo. Igual que hicimos para escribir un fichero, podríamos construir una caja de diálogo y preguntarle al usuario por el nombre del fichero a leer. Pero esto no es práctico, sobre todo cuando el sistema de ficheros es grande. Nosotros tenemos que ser capaces de buscar un fichero por todo el disco, de la misma forma que lo hace Windows, esto es, elegimos en una ventana la unidad de disco, después en otra elegimos el directorio, y por último en la ventana de ficheros elegimos el nombre del fichero que deseamos leer. Para realizar estas operaciones Visual Basic tiene disponibles en el panel de utilidades tres controles: *lista de unidades de disco*, *lista de directorios* y *lista de ficheros* (vea el panel de utilidades en el Capítulo 2). Por ejemplo, cree una forma y asígnela el título *Sistema de Ficheros*, el nombre *SistemaF*, e incluya en la misma los siguientes controles:

	Caption	CtlName
Etiqueta 1	Ficheros:	Label1
Lista de ficheros		File1
Etiqueta 2	Directorios:	Label2
Lista de directorios		Dir1
Etiqueta 3	Unidades:	Label3
Lista de unidades		Drive1

El resultado, es la forma que se presenta a continuación.



La propiedad **Drive** del control *lista de unidades de disco* permite saber cuál es la unidad actual. Cuando queremos modificar la unidad actual, sólo el primer carácter de la cadena de caracteres correspondiente, es significativo. Un cambio de unidad de disco, genera el suceso **Change**. Esta propiedad sólo está disponible en tiempo de ejecución. Por ejemplo,

```
SistemaF.Drive1.Drive = "d:"
```

El directorio actual en el control *lista de directorios* aparece sombreado. La propiedad **Path** devuelve el camino completo del directorio actual, incluyendo el nombre de la unidad. Modificando el valor de esta propiedad podemos cambiar el directorio actual. Si sólo modificamos la unidad (por ejemplo *d:*), por defecto se selecciona el directorio actual en dicha unidad. Un cambio de directorio actual, genera el suceso **Change**. Esta propiedad sólo está disponible en tiempo de ejecución y pertenece también al control *lista de ficheros*, con la diferencia de que aquí un cambio del camino actual, genera el suceso **PathChange**. Por ejemplo,

```
SistemaF.Dir1.Path = "d: temp"
```

La propiedad **Pattern** del control *lista de ficheros* permite que se visualicen solamente los ficheros que cumplan ese patrón. Por ejemplo, **.mak* hace que se visualicen sólo los ficheros con extensión *.mak*. La propiedad **FileName** permite especificar el fichero que se quiere leer, o devuelve el nombre del fichero seleccionado en la lista; esta propiedad sólo está disponible en tiempo de ejecución. Por ejemplo,

```
NombreF$ = SistemaF.File1.FileName
```

Utilización conjunta de estos controles

Para sincronizar los controles *Drive1*, *Dir1* y *File1* de la forma que acabamos de diseñar, la secuencia de pasos a seguir es la siguiente:

1. El usuario selecciona una unidad de disco de la lista.
2. En el control *Drive1*, se visualiza la nueva unidad y se genera el suceso **Change** para este control.
3. Para que se visualice simultáneamente la lista de directorios correspondiente a la nueva unidad seleccionada, el procedimiento *Drive1_Change* tiene que asignar la nueva unidad a la propiedad **Path** del control *Dir1*.

```
Sub Drive1_Change
  Dir1.Path = Drive1.Drive
End Sub
```


4. En el control *Dir1*, se visualiza la nueva lista de directorios y se genera el suceso **Change** para éste control.
5. Para que se visualice simultáneamente la lista de ficheros correspondiente al directorio actual, el procedimiento *Dir1_Change* tiene que asignar el nuevo camino a la propiedad **Path** del control *File1*.

```
Sub Dir1_Change ()
  File1.Path = Dir1.Path
End Sub
```

6. En el control *File1*, se visualiza la nueva lista de ficheros y se genera el suceso **PathChange** para éste control.

Escriba estos procedimientos y continuemos con nuestra aplicación. Pensemos qué debe suceder cuando el usuario haga clic en la orden *Abrir* del menú *Fichero* de nuestra aplicación *Editor*. La respuesta es sencilla, se tiene que mostrar la ventana *Sistema de Ficheros*. Para realizar esto, abra la ventana de código correspondiente a la orden *Abrir* y escriba el procedimiento que se muestra a continuación.

```
Sub FicheroAbrir_Click ()
  SistemaF.Show
End Sub
```

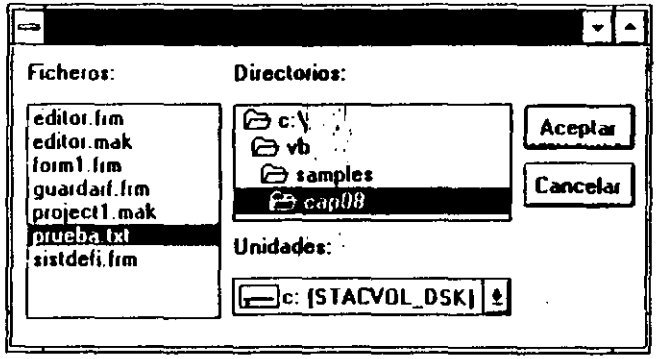
Guarde la aplicación y a continuación ejecútela. Observe como al cambiar de unidad se visualiza la lista de directorios correspondiente. Seleccione uno de estos directorios y observe que haciendo un doble clic sobre el directorio seleccionado, aparece la lista de ficheros correspondiente al mismo.

ABRIR UN FICHERO PARA LEER

Una vez presentado el sistema de ficheros, el usuario seleccionará del mismo el fichero que desea abrir, haciendo clic sobre su nombre. A continuación, para cargarlo, pulsará el botón *Aceptar* o hará un doble clic sobre el nombre del mismo. En otro caso, si desea cancelar la operación, pulsará el botón *Cancelar*.

De acuerdo con lo expuesto, primeramente añada a la forma *Sistema de Ficheros* los botones *Aceptar* y *Cancelar* y asígneles un nombre igual al título. La propiedad **Default** del botón *Aceptar* póngala a valor **True**; esto hará que cuando pulse la tecla **Enter** el resultado sea el mismo que si hubiera pulsado el botón *Aceptar*. A continuación, escriba el código para que estos botones actúen como deseamos. El código para el botón *Cancelar* es el más sencillo, pues simplemente se trata de ocultar la forma. Esto es,

```
Sub Cancelar_Click ()
  SistemaF.Hide
End Sub
```



Escribamos ahora el código para el procedimiento *Aceptar_Click*. Este botón va a tener múltiples funciones, esto es, aceptar una unidad, aceptar un directorio o aceptar un fichero seleccionado.

Cuando el usuario hace clic sobre la unidad que desea seleccionar en el control *Drive1*, se genera el suceso **Change** para este control; también se genera este suceso cuando después de haber hecho la selección con las teclas de dirección, el control se desenfoca manualmente o en ejecución (**SetFocus**).

Para aceptar un directorio o aceptar un fichero seleccionado, necesitaremos primero saber de qué elemento se trata. Para ello defina a nivel de la forma *Sistema de Ficheros* (seleccione **(general)**), las siguientes constantes y variables que a continuación vamos a utilizar.

```
Const CLIC_EN_DIR = 1, CLIC_EN_FILE = 2
Dim ÚltimoCambio As Integer
```

Cuando el usuario pulse el botón *Aceptar*, o la tecla **Enter**, el procedimiento asociado tendrá que verificar la variable *ÚltimoCambio* para saber si lo que hay que aceptar es un directorio o un fichero. El esquema de este procedimiento es el siguiente:

```
Sub Aceptar_Click ()
  Aceptar.SetFocus
  Select Case ÚltimoCambio
    Case CLIC_EN_FILE
      ...
    Case CLIC_EN_DIR
      ...
  End Select
  ...
End Sub
```


La sentencia *Aceptar.SetFocus* enfoca el botón *Aceptar*. Esta operación es necesaria después de pulsar **Enter** para aceptar la unidad de disco, si ésta fue seleccionada con las teclas de dirección.

Cuando se genera el suceso **Click** para el control *Dir1* significa que el usuario ha seleccionado, pero no ha aceptado, un directorio. Esta selección puede realizarla haciendo clic sobre el directorio o con las teclas de dirección. La selección realizada, la confirmará pulsando el botón *Aceptar* o haciendo un doble clic sobre el directorio seleccionado. Entonces, en el procedimiento *Dir1_Click* no tomaremos ninguna acción, sino simplemente anotaremos que se ha dado este hecho.

```
Sub Dir1_Click ()
  ÚltimoCambio = CLIC_EN_DIR
End Sub
```

Si después de haber seleccionado un directorio el usuario pulsa el botón *Aceptar*, este será ahora el directorio actual, se genera el suceso **Change** para este control y por lo tanto, se actualiza la ventana de ficheros. El procedimiento *Aceptar_Click* será ahora así:

```
Sub Aceptar_Click ()
  Aceptar.SetFocus
  Select Case ÚltimoCambio
    Case CLIC_EN_DIR
      Dir1.Path = Dir1.List(Dir1.ListIndex)
    Case CLIC_EN_FILE
      ...
  End Select
  ÚltimoCambio = 0
End Sub
```

La expresión *Dir1.List(Dir1.ListIndex)* devuelve el camino del directorio actualmente seleccionado (vea *listas* y *combinados* en el Capítulo 7). Esta operación está implícita cuando la aceptación se hace con un doble clic.

Cuando se genera el suceso **Click** para el control *File1* significa que el usuario ha seleccionado, pero no ha aceptado, un fichero. Esta selección puede realizarla haciendo clic sobre el fichero o con las teclas de dirección. La selección realizada la confirmará pulsando el botón *Aceptar* o haciendo un doble clic sobre

el fichero seleccionado. Entonces en el procedimiento *File1_Click* no tomaremos ninguna acción, sino simplemente anotaremos que se ha dado esto hecho.

```
Sub File1_Click
  ÚltimoCambio = CLIC_EN_FILE
End Sub
```

Cuando el usuario hace un doble clic sobre el fichero seleccionado para cargarlo, el resultado es equivalente a seleccionar el fichero y pulsar el botón *Aceptar*. Por lo tanto, el procedimiento *File1_DblClick* tiene que contener las siguientes sentencias:

```
Sub File1_DblClick ()
  ÚltimoCambio = CLIC_EN_FILE
  Aceptar_Click
End Sub
```

Si después de haber seleccionado un fichero el usuario pulsa el botón *Aceptar*, el procedimiento *Aceptar_Click* reconocerá que se ha aceptado un fichero, tomará su nombre y lo abrirá para leer.

Leer de un fichero secuencial

Para realizar el proceso de leer la información de un fichero secuencial, hay que realizar tres pasos:

1. *Abrir el fichero para leer* **(Input)** utilizando la sentencia **Open**. La sintaxis de esta sentencia es:

   Open *nombre-fich* **[For** *modo***] As [#]***nº fichero*

   nombre-fich es el nombre del fichero (puede incluir el camino);
   modo puede ser Output, Input, Append, Binary, o Random;
   nº fichero es un número entre 1 y 255 para referenciar el fichero.

2. *Leer los datos del fichero* utilizando la sentencia **Input #** o **InputS**. En nuestro caso utilizaremos **InputS** cuya sintaxis es:

   vS = **InputS** (*n*, **#***nº fichero*)

   n es el número de caracteres a leer;
   nº fichero es el número con el que se abrió el fichero para leer.

3. *Cerrar el fichero* mediante la sentencia **Close**. La sintaxis de esta sentencia es:

   Close [[#]*nº fichero* [,[#]*nº fichero*]...]

nº fichero es el número con el que se abrió el fichero.

Continuemos con la aplicación. Abra la ventana de código correspondiente al botón *Aceptar* de la forma *Sistema de Ficheros* y complete el procedimiento *Aceptar_Click* como se indica a continuación:

```
Sub Aceptar_Click ()
  Dim Fichero As String
  Aceptar.SetFocus
  Select Case ÚltimoCambio
    Case CLIC_EN_DIR
      Dir1.Path = Dir1.List(Dir1.ListIndex)
    Case CLIC_EN_FILE
      If (Right$(Dir1.Path, 1) = "\") Then
        Fichero = Dir1.Path + File1.FileName
      Else
        Fichero = Dir1.Path + "\" + File1.FileName
      End If
      Open Fichero For Input As #1
      Form1.Text1.Text = Input$(LOF(1), #1)
      Close #1
      SistemaF.Hide
  End Select
  ÚltimoCambio = 0
End Sub
```

Este procedimiento tiene dos partes: buscar el nombre del fichero y abrir este fichero para leer.

El nombre del fichero a abrir se almacena en la variable *Fichero*. La expresión *Dir1.Path* representa el camino completo incluyendo la unidad de disco (por ejemplo, *c:\vb\samples*) y nosotros añadimos el carácter "\" y el nombre del fichero dado por la expresión *File1.FileName*. Esto nos permitirá recuperar cualquier fichero de texto de cualquier directorio. Sin embargo, cuando el camino *Dir1.Path* se corresponde con el directorio raíz, tal como *d:*, obtendríamos un carácter "\" de más (por ejemplo, *d:\\fichero.txt*). Para evitar esto, chequearemos el último carácter de *Dir1.Path*. El resultado es:

```
If (Right$(Dir1.Path, 1) = "\") Then
  Fichero = Dir1.Path + File1.FileName
Else
  Fichero = Dir1.Path + "\" + File1.FileName
End If
```

La sentencia **Open** abre el fichero referenciado por *Fichero* para leer **(Input)** y le asigna el número 1. Si el fichero no es de texto o es demasiado grande

(mayor de 64 Kb), se producirá un error. Para detectar los posibles errores y permitir la continuación del programa, modificaremos el procedimiento *Aceptar_Click* de la forma siguiente:

```
Sub Aceptar_Click ()
  Dim Fichero As String
  Aceptar.SetFocus
  Select Case ÚltimoCambio
    Case CLIC_EN_DIR
      Dir1.Path = Dir1.List(Dir1.ListIndex)
    Case CLIC_EN_FILE
      On Error GoTo RutinaError
      If (Right$(Dir1.Path, 1) = "\") Then
        Fichero = Dir1.Path + File1.FileName
      Else
        Fichero = Dir1.Path + "\" + File1.FileName
      End If
      Open Fichero For Input As #1
      Form1.Text1.Text = Input$(LOF(1), #1)
      Close #1
      SistemaF.Hide
  End Select
Salir:
  ÚltimoCambio = 0
  Exit Sub
RutinaError:
  MsgBox "Error: no se puede abrir el fichero", 48, "Editor"
  Close
  Resume Salir
End Sub
```

También puede ocurrir un error al seleccionar la unidad de disco (por ejemplo, la unidad *a:*) debido a que ésta no esté preparada. Para detectar este posible error, modificaremos el procedimiento *Drive1_Change* de la forma siguiente:

```
Sub Drive1_Change ()
  On Error GoTo Driver
  Dir1.Path = Drive1.Drive
  Exit Sub
Driver:
  MsgBox "Error: unidad no preparada", 48, "Editor"
  Exit Sub
End Sub
```

La función **Input$** lee del fichero número 1 un número de caracteres igual a la longitud del mismo (**LOF(1)**) y almacena la información en *Form1.Text1.Text* que corresponde con el área de edición. Finalmente la sentencia **Close #1** cierra el fichero número 1. Un fichero abierto no puede ser vuelto a abrir en ninguno de los modos: para volverlo a abrir en el mismo u otro modo, con igual o diferente número, hay que cerrarlo previamente.

La función **Input$** está limitada a leer 32767 caracteres de un fichero abierto en modo secuencial o binario. Para leer ficheros más largos, simplemente hay que chequear la longitud del fichero y leerlo en varias veces como se especifica a continuación.

```
Dim Longitud As Long, N As Integer

Longitud = LOF(1)
Texto = ""
Do
  If Longitud > 32767 Then
    N = 32767
  Else
    N = Longitud
  End If
  Texto = Texto + Input$(N, #1)
  Longitud = Longitud - 32767
Loop Until Longitud <= 0
Form1.Text1.Text = Texto
```

En el ejemplo anterior, la variable *Texto* es de tipo **String** y está definida a nivel de la forma. Si define esta variable como local, tendrá problemas de espacio de memoria.

Similarmente, la sentencia **Line Input #** lee de un fichero cadenas de caracteres delimitadas por retornos de carro (el delimitador no se lee). Esto significa que otra forma de leer un fichero, es utilizando esta sentencia como se especifica a continuación. En cualquier caso, y para nuestra aplicación, esta forma resulta muy lenta, por lo que no es aconsejable.

```
Dim Línea As String

Do
  Line Input #1, Línea
  Form1.Text1.Text = Form1.Text1.Text + Línea + Chr$(13) + Chr$(10)
Loop Until EOF(1)
```

Observe que por cada *Línea* de texto que añadimos a la caja de texto hay que añadir también el retorno de carro que no se lee (*Chr$(13) + Chr$(10)*).

La función **EOF** retorna un valor de –1 (**True**) si se ha alcanzado el final del fichero especificado, en nuestro caso el fichero 1, y un valor 0 (**False**) en caso contrario.

Recuerde que la propiedad **Text** es de tipo **String** y por lo tanto está limitada a aproximadamente 65535 caracteres.

UTILIZACIÓN DE FICHEROS ALEATORIOS

Vamos a añadir soporte a nuestra aplicación *Libros*, para lo que añadiremos al menú *Fichero* las órdenes de *Abrir* y *Guardar*.

Cargue la aplicación *Libros* que realizó en el Capítulo 7 y añada al menú *Fichero* las órdenes con las propiedades que se indican en la tabla siguiente.

Caption	CtlName	Accelerator
Abrir...	FicheroAbrir	Ctrl+B
Guardar...	FicheroGuardar	Ctrl+G

Recuerde que nuestra base de datos estaba formada por un array de registros denominado *Libros*, definido en el módulo global *libros.bas* como se indica a continuación.

```
Type Registro
  Titulo As String * 30
  Autor As String * 30
  Editorial As String * 12
  Prestado As String * 240
End Type
Global Libros(1 To 40) As Registro
```

Escribir en un fichero aleatorio

Cuando el usuario haga clic en la orden *Guardar* del menú *Fichero*, es porque quiere guardar la base de datos *Libros* en un fichero. Para realizar esta operación, podemos utilizar la misma caja de diálogo que en la aplicación anterior. Para cargar la forma *Guardar Fichero* de la aplicación *Editor* en la aplicación *Libros*, seleccione la orden **Add File** (Añadir Fichero) del menú **File** de Visual Basic y seleccione el fichero *frm* correspondiente de la lista de ficheros.

A continuación visualice la ventana de código correspondiente a la orden *Guardar* del menú *Fichero* y escriba:


```
Sub FicheroGuardar_Click ()
  GuardarF.Show
End Sub
```

Modifiquemos ahora los procedimientos asociados con esta forma para adaptarlos a la aplicación *Libros*. Así, el procedimiento *Cancelar_Click* asociado al botón *Cancelar*, cuando finalice, tiene ahora que enfocar a la caja *Título* de la forma *Form1*. Realizado este cambio, el procedimiento queda como sigue:

```
Sub Cancelar_Click ()
  NombreFg.Text = ""
  NombreFg.SetFocus
  GuardarF.Hide
  Form1.Titulo.SetFocus
End Sub
```

El procedimiento *Aceptar_Click* asociado con el botón *Aceptar*, permite guardar la información requerida en un fichero de nombre, el escrito en la caja de texto *NombreFg*. Para guardar los registros de la base de datos utilizaremos ahora, un fichero aleatorio en lugar de un fichero secuencial. Para escribir en un fichero aleatorio, hay que realizar tres pasos:

1. *Abrir el fichero para acceso al azar* utilizando la sentencia **Open**. La sintaxis en este caso es la siguiente:

   Open *nombre-fich* **[For Random] As [#]***nº fichero* **Len =** *longitud reg*

   nombre-fich es el nombre del fichero (puede incluir el camino);
   nº fichero es un número entre 1 y 255 para referenciar el fichero;
   longitud reg es una expresión entera que indica la longitud de cada uno de los registros de un fichero aleatorio.

2. *Escribir los datos en el fichero* utilizando la sentencia **Put**. La sintaxis de esta sentencia es:

   Put [#]*nº fichero*, [*nº registro*][, *variable*]

   nº fichero es el número con el que se abrió el fichero;
   nº registro es el número 1, 2, 3, ..., del registro. Si se omite se toma el número siguiente al utilizado en la última sentencia **Put**;
   variable contiene los datos a escribir en el fichero.

3. *Cerrar el fichero* una vez finalizadas las operaciones de E/S mediante la sentencia **Close**.

Cuando se abre un fichero en modo **Random** si no existe se crea y si existe simplemente se abre. En cualquier caso, el fichero queda abierto para leer y es-

cribir a menos que se especifique otra cosa mediante la cláusula Access. Escribir encima de cualquier registro del fichero significa modificar ese registro.

Para escribir la base de datos, ahora almacenada en el array *Libros*, en el fichero referenciado por *NombreFg.Text*, las operaciones son las siguientes:

```
Open NombreFg.Text For Random As #1 Len = Len(Libros(1))
For I = 1 To TotalRegs
  Put #1, , Libros(I)
Next I
Close #1
```

En el supuesto de que el fichero ya exista, el proceso anterior sobrescribirá los registros existentes a partir del 1. Cuando se modifique la base de datos y se guarde una y otra vez en el fichero, sucederá que si el número de registros del array *Libros* es menor que el número actual de registros del fichero, los registros en demasía quedarán, cuando la realidad es que tienen que desaparecer. Una forma fácil de dar solución a este problema es verificar antes de escribir en el fichero si éste existe, en cuyo caso lo borramos y lo creamos de nuevo. Las sentencias siguientes realizan este proceso.

```
Open NombreFg.Text For Random As #1 Len = Len(Libros(1))
If (LOF(1) <> 0) Then
  Cadena = "El fichero ya existe" + Chr$(13) + Chr$(10)
  Cadena = Cadena + "¿desea sobrescribirlo?"
  If MsgBox(Cadena, 36, "Libros") = 6 Then
    Close #1
    Kill NombreFg.Text
    Open NombreFg.Text For Random As #1 Len = Len(Libros(1))
  Else
    Close #1
    NombreFg.SetFocus
    Exit Sub
  End If
End If
```

En el caso de que el fichero exista (su longitud no es 0), la función **MsgBox** visualizará el mensaje dado por *Cadena* en una forma con un icono "?" y con dos botones *Sí* y *No* (icono (32) + botones *SiNo* (4) = 36). Si se pulsa el botón *Sí* la función devuelve el valor 6 y si se pulsa el botón *No* devuelve el valor 7. Observe que el mensaje escrito por la función **MsgBox** consta de dos líneas.

Si el usuario pulsa el botón *Sí*, la sentencia **Kill** borra el fichero especificado, el cual tiene que estar cerrado, y la sentencia **Open** crea de nuevo el fichero; si pulsa el botón *No*, se enfoca de nuevo la caja de texto, para permitir al usuario modificar el nombre del fichero, y se abandona este procedimiento.

Abra la ventana de código correspondiente al botón *Aceptar* de la forma *Guardar Fichero* y modifique el procedimiento *Aceptar_Click* para que realice las operaciones indicadas.

```
Sub Aceptar_Click ()
  Dim I As Integer, Cadena As String
  On Error GoTo RutinaDeError
  'Escribir en el fichero
  Open NombreFg.Text For Random As #1 Len = Len(Libros(1))
  If (LOF(1) <> 0) Then
    Cadena = "El fichero ya existe" + Chr$(13) + Chr$(10)
    Cadena = Cadena + "¿desea sobrescribirlo?"
    If MsgBox(Cadena, 36, "Libros") = 6 Then
      Close #1
      Kill NombreFg.Text
      Open NombreFg.Text For Random As #1 Len = Len(Libros(1))
    Else
      Close #1
      NombreFg.SetFocus
      Exit Sub
    End If
  End If
  For I = 1 To TotalRegs
    Put #1, , Libros(I)
  Next I
  Close #1
  'Ocultar la forma
  NombreFg.Text = ""
  NombreFg.SetFocus
  GuardarF.Hide
  Form1.Titulo.SetFocus
Salir:
  Exit Sub
RutinaDeError:
  MsgBox "Error al abrir el fichero", 48, "Libros"
  NombreFg.SetFocus
  Resume Salir
End Sub
```

Guarde la forma que acaba de modificar con otro nombre, para no cambiar el fichero *.frm* de la aplicación *Editor*. Para ello ejecute la orden **Save File As** del menú **File** de Visual Basic.

Leer de un fichero aleatorio

Cuando el usuario haga clic en la orden *Abrir* del menú *Fichero* es porque quiere leer la base de datos almacenada en un fichero conocido y almacenarla en el array *Libros*. Para realizar esta operación, podemos utilizar la misma caja de diálogo *Sistema de Ficheros* que en la aplicación anterior. Para cargar en nuestra aplicación *Libros* la forma que da lugar a esta caja de diálogo, seleccione la orden **Add File** (Añadir Fichero) del menú **File** de Visual Basic y seleccione el fichero *frm* correspondiente de la lista de ficheros.

A continuación visualice la ventana de código correspondiente a la orden *Abrir* del menú *Fichero* y escriba:

```
Sub FicheroAbrir_Click ()
  SistemaF.Show
End Sub
```

Modifiquemos ahora los procedimientos asociados con esta forma para adaptarlos a la aplicación *Libros*. El único procedimiento que cambia es *Aceptar_Click*, que se tiene que encargar de leer la base de datos del fichero y almacenarla en el array *Libros*. Para leer los registros de la base de datos, teniendo en cuenta que el fichero es aleatorio, hay que realizar tres pasos:

1. *Abrir el fichero para acceso al azar* utilizando la sentencia **Open**.
2. *Leer el registro deseado del fichero* utilizando la sentencia **Get**. La sintaxis de esta sentencia es la siguiente:

   Get [#]*nº fichero*, [,[*nº registro*][, *variable*]

   nº fichero es el número con el que se abrió el fichero:
   nº registro es el número 1, 2, 3, ..., del registro. Si se omite se toma el número siguiente al utilizado en la última sentencia **Get**:
   variable almacena los datos del registro leído del fichero.
3. *Cerrar el fichero* una vez finalizadas las operaciones de E/S mediante la sentencia **Close**.

Para leer la base de datos del fichero referenciado por *NombreFg.Text* y almacenarla en el array *Libros*, las operaciones son las siguientes:

```
Open Fichero For Random As #1 Len = LongReg
NumRegsFich = LOF(1) \ LongReg
For I = 1 To NumRegsFich
  Get #1, , Libros(I)
Next I
Close #1
```


La operación de leer el fichero y cargar el array no activa la base de datos. Por eso, a continuación, tenemos que añadir a la lista clasificada *ListaLibros*, los títulos de cada uno de los registros del array, con el fin de que el usuario pueda seleccionar de la misma cualquiera de ellos. Para realizar esta operación, es necesario previamente borrar todas las entradas actuales de la lista clasificada. De no hacerlo así, la carga de una base de datos se añadiría a la anteriormente cargada. Finalmente, se visualiza en la caja de diálogo titulada *Libros Prestados* y denominada *Form1*, el primer registro. Estas operaciones, se realizan de la forma siguiente:

```
'Borrar los registros existentes en la base de datos
For I = 1 To TotalRegs
  Form2.ListaLibros.RemoveItem 0
Next I
TotalRegs = NumRegsFich

'Añadir los elementos del array a la lista
For I = 1 To TotalRegs
  Form2.ListaLibros.AddItem Libros(I).Título
Next I

'Visualizar el primer registro
Form1.Título.Text = Libros(1).Título
Form1.Autor.Text = Libros(1).Autor
Form1.Editorial.Text = Libros(1).Editorial
Form1.Prestado.Text = Libros(1).Prestado
```

Abra la ventana de código correspondiente al botón *Aceptar* de la forma *Sistema de Ficheros* y modifique el procedimiento *Aceptar_Click* para que realice las operaciones indicadas.

```
Sub AceptarClick ()
  Dim Fichero As String
  Dim NumRegsFich As Integer, LongReg As Integer, I As Integer
  LongReg = Len(Libros(1))
  Aceptar.SetFocus
  Select Case ÚltimoCambio
    Case CLIC_EN_DIR
      Dir1.Path = Dir1.List(Dir1.ListIndex)

    Case CLIC_EN_FILE
      On Error GoTo RutinaError
      If (Right$(Dir1.Path, 1) = "\") Then
        Fichero = Dir1.Path + File1.FileName
```

```
      Else
        Fichero = Dir1.Path + "\" + File1.FileName
      End If
      Open Fichero For Random As #1 Len = LongReg
      NumRegsFich = LOF(1) \ LongReg
      'Leer los registros del fichero
      For I = 1 To NumRegsFich
        Get #1, , Libros(I)
      Next I
      Close #1
      'Borrar los registros existentes en la base de datos
      For I = 1 To TotalRegs
        Form2.ListaLibros.RemoveItem 0
      Next I
      TotalRegs = NumRegsFich
      'Añadir los elementos del array a la lista
      For I = 1 To TotalRegs
        Form2.ListaLibros.AddItem Libros(I).Título
      Next I
      'Visualizar el primer registro
      Form1.Título.Text = Libros(1).Título
      Form1.Autor.Text = Libros(1).Autor
      Form1.Editorial.Text = Libros(1).Editorial
      Form1.Prestado.Text = Libros(1).Prestado
      SistemaF.Hide
  End Select
Salir:
  ÚltimoCambio = 0
  Exit Sub
RutinaError:
  MsgBox "Error: no se puede abrir el fichero", 48, "Libros"
  Close
  Resume Salir
End Sub
```

Trabajando directamente sobre el fichero

El array *Libros* ocupa una cantidad de memoria considerable y dependiendo del tamaño de la base de datos puede incluso que no podamos crearlo. Esto nos conduce a pensar como prescindir del array y trabajar directamente sobre el fichero en disco, aunque esta forma de trabajo resulte un poco más lenta.

En primer lugar defina *Libros* como un registro y no como un array y añada la variable *Fichero* como global. *Fichero* especifica la base de datos sobre la que está trabajando.


```
Type Registro
  Título As String * 30
  Autor As String * 30
  Editorial As String * 12
  Prestado As String * 240
End Type
Global Libros As Registro
Global TotalRegs As Integer
Global Fichero As String
```

Esto implica que el procedimiento *VisualizarRegistro* cargue el registro seleccionado de la lista, directamente del fichero.

```
Sub VisualizarRegistro ()
  Dim I As Integer, Encontrado As Integer
  Encontrado = 0 'elemento no encontrado
  'Buscar el registro seleccionado
  Open Fichero For Random As #1 Len = Len(Libros)
  For I = 1 To LOF(1) \ Len(Libros)
    Get #1, I, Libros
    If (RTrim$(Libros.Título) = RTrim$(Form2.ListaLibros.Text)) Then
      Encontrado = -1 'elemento encontrado
      Exit For
    End If
  Next I
  Close #1
  If (Encontrado) Then
    'Visualizar registro
    Form1.Título.Text = Libros.Título
    Form1.Autor.Text = Libros.Autor
    Form1.Editorial.Text = Libros.Editorial
    Form1.Prestado.Text = Libros.Prestado
    Form2.Hide
  Else
    MsgBox "El libro especificado no se encuentra", 48, "Libros"
  End If
End Sub
```

El procedimiento *Aceptar_Click* asociado con el botón *Aceptar* de la forma *Sistema de Ficheros*, se modifica en el sentido de que ahora no carga un array con los registros del fichero, y en que la lista clasificada, *ListaLibros*, se crea leyendo cada registro directamente del fichero. La variable *Fichero* se ha definido como global.

```
Sub Aceptar_Click ()
  Dim NumRegsFich As Integer, LongReg As Integer, I As Integer
  LongReg = Len(Libros)
  Aceptar.SetFocus
```

```
  Select Case UltimoCambio
    Case CLIC_EN_DIR
      Dir1.Path = Dir1.List(Dir1.ListIndex)
    Case CLIC_EN_FILE
      On Error GoTo RutinaError
      If (Right$(Dir1.Path, 1) = "\") Then
        Fichero = Dir1.Path + File1.FileName
      Else
        Fichero = Dir1.Path + "\" + File1.FileName
      End If
      Close
      Open Fichero For Random As #1 Len = LongReg
      NumRegsFich = LOF(1) \ LongReg
      'Borrar los registros existentes en la base de datos
      For I = 1 To TotalRegs
        Form2.ListaLibros.RemoveItem 0
      Next I
      TotalRegs = NumRegsFich
      'Añadir los elementos del array a la lista
      For I = 1 To TotalRegs
        Get #1, I, Libros
        Form2.ListaLibros.AddItem Libros.Título
      Next I
      'Visualizar el primer registro
      Get #1, 1, Libros
      Form1.Título.Text = Libros.Título
      Form1.Autor.Text = Libros.Autor
      Form1.Editorial.Text = Libros.Editorial
      Form1.Prestado.Text = Libros.Prestado
      Close #1
      SistemaF.Hide
  End Select
Salir:
  ÚltimoCambio = 0
  Exit Sub
RutinaError:
  MsgBox "Error: no se puede abrir el fichero", 48, "Libros"
  Close
  Resume Salir
End Sub
```

Modifique el procedimiento *FicheroGuardar_Click* asociado con la orden *Guardar* del menú *Fichero*, como se indica a continuación. Este procedimiento visualiza la forma *GuardarF* y el nombre del fichero actual en su caja de texto.

```
Sub FicheroGuardar_Click ()
  GuardarF.Show
  GuardarF.NombreFg.Text = Fichero
End Sub
```


El procedimiento *Aceptar_Click* asociado con el botón *Aceptar* de la forma *Guardar Fichero*, puesto que ya no existe el array *Libros*, se modifica de la forma siguiente: como es posible que hayamos borrado elementos de la lista, debemos crear un nuevo fichero, con el nombre especificado por el usuario, que contenga sólo los registros del fichero viejo (*Fichero*) que pertenezcan a la lista. Para ello, primero renombramos *Fichero* con el nombre *Temp*; a continuación abrimos el fichero especificado por el usuario para guardar la base de datos, y copiamos en éste todos los registros de *Temp* que estén en la lista de libros (*ListaLibros*).

Este proceso resulta satisfactorio si no hay títulos repetidos. Dejo para el lector el tratamiento de este problema, si lo cree necesario.

La sentencia **Name** permite cambiar el nombre de un fichero. Si al escribir el nombre nuevo se indica un camino diferente al camino original, el fichero, además de ser renombrado, es movido al nuevo directorio especificado.

```
Sub Aceptar_Click ()
  Dim I As Integer, Cadena As String
  On Error GoTo RutinaDeError
  Name Fichero As "Temp" 'si Fichero no existe, error 53
  'Escribir en el fichero
  Open NombreFg.Text For Random As #1 Len = Len(Libros)
  If (LOF(1) <> 0) Then
    Cadena = "El fichero ya existe" + Chr$(13) + Chr$(10)
    Cadena = Cadena + "¿desea sobrescribirlo?"
    If MsgBox(Cadena, 36, "Libros") = 6 Then
      Close #1
      Kill NombreFg.Text
      Open NombreFg.Text For Random As #1 Len = Len(Libros)
    Else
      Close #1
      NombreFg.SetFocus
      Exit Sub
    End If
  End If
  Open "Temp" For Random As #2 Len = Len(Libros)
  For R = 1 To LOF(2) / Len(Libros)
   Get #2, , Libros
   For I = 0 To TotalRegs - 1
    If (RTrim$(Libros.Título)= RTrim$(Form2.ListaLibros.List(I))) Then
      Put #1, , Libros
      Exit For
    End If
   Next I
  Next R
  Close #1, #2
  Kill "Temp"
```

```
  Fichero = NombreFg.Text 'nuevo fichero creado
  'Ocultar la forma
  NombreFg.Text = ""
  NombreFg.SetFocus
  GuardarF.Hide
  Form1.Título.SetFocus
Salir:
  Exit Sub
RutinaDeError:
  Select Case Err
    Case 53 'el fichero no existe
      Resume Next
    Case Else
      MsgBox "Error al abrir el fichero", 48, "Libros"
  End Select
  NombreFg.SetFocus
  Resume Salir
End Sub
```

En el procedimiento *Borrar_Click* asociado con el botón *Borrar* de la forma *Buscar Registro*, desaparece la parte que borraba un registro del array *Libros*, quedando dicho procedimiento como sigue:

```
Sub Borrar_Click ()
  Dim R As Integer
  'Borrar el registro de la lista
  'Posición en la lista
  R = Form2.ListaLibros.ListIndex
  'Se borra
  Form2.ListaLibros.RemoveItem R
  TotalRegs = Form2.ListaLibros.ListCount
  Form2.Hide
End Sub
```

El procedimiento *AñadirReg_Click* asociado con la orden *Añadir Registro* del menú *Fichero*, ahora debe añadir el registro al final del fichero, y el título a la lista de libros.

```
Sub AñadirReg_Click ()
  Dim          As Integer
  Libros.      tulo = Form1.Título.Text
  Libros.Autor = Form1.Autor.Text
  Libros.Editorial = Form1.Editorial.Text
  Libros.Prestado = Form1.Prestado.Text
  Open Fichero For Random As #1 Len = Len(Libros)
  NumRegs = LOF(1)  Len(Libros)
  '    #1, NumRegs + 1, Libros
  C      #1
  Fo.  .ListaLibros.AddItem Título.Text
```



```
  TotalRegs = TotalRegs + 1
End Sub
```

Cuando el usuario desee crear un fichero nuevo, simplemente añadirá registros y después los guardará en el fichero deseado, ejecutando la orden *Guardar* del menú *Fichero*. Esto exige inicializar la variable *Fichero* a un valor, operación que realiza el procedimiento *Form_Load*. Según esto, el procedimiento *AñadirReg_Clik* añadirá los registros al fichero denominado *Temporal* y el procedimiento *Aceptar_Click* asociado con el botón *Aceptar* de la forma *Guardar Fichero*, lo copiará sobre el nuevo fichero especificado.

```
Sub Form_Load ()
  Fichero = "Temporal"
End Sub
```

SENTENCIA Seek

La sentencia **Seek** permite situar la posición de lectura o de escritura en una posición determinada dentro del fichero. Su sintaxis es:

Seek [#]*nº fichero, posición*

Por ejemplo, la sentencia:

```
Get #1, I, Libros(I)
```

es equivalente a las sentencias:

```
Seek #1, I
Get #1, , Libros(I)
```

La sentencia **Get** lee tantos bytes como quepan en la variable y la sentencia **Put** escribe tantos bytes como haya en la variable.

El argumento *posición* es un entero largo y especifica la nueva posición dentro del fichero. Para un fichero secuencial o binario, *posición* es interpretado como el número de byte (carácter) a partir del cual queremos leer o escribir dentro del fichero; para ficheros aleatorios, *posición* es interpretado como el número de registro a partir del cual queremos leer o escribir dentro del fichero.

Esta sentencia nos permite trabajar byte a byte en ficheros binarios. Por ejemplo:

```
Dim a As String * 5
```

```
Open "Datos.com" For Binary As #1
Seek #1, 11
Get #1, , a
Seek #1, 1
Put #1, , a
```

Partiendo de que la primera posición dentro del fichero es la 1, este ejemplo almacena en *a* los caracteres de las posiciones 11, 12, 13, 14 y 15, quedando en 16 la posición actual dentro del fichero. A continuación, se escribe el contenido de *a* en las posiciones 1, 2, 3, 4 y 5, quedando en 6 la posición actual.

FUNCIONES Loc y LOF

Recordamos por el interés que tienen, que las funciones **Loc**(*n*) y **LOF**(*n*) dan como resultado la posición actual en el fichero y la longitud o número de caracteres del fichero *n* especificado.

Para un fichero secuencial, **Loc** devuelve el número de bloques de 128 bytes leídos o escritos; para un fichero aleatorio, devuelve el número del último registro leído o escrito; y para un fichero binario, devuelve la posición del último byte leído o escrito.

Para un fichero aleatorio, el valor devuelto por la función **LOF** dividido por la longitud del registro (argumento **Len** o 128 por defecto) da el número de registros del fichero. Para un fichero secuencial en el que los registros son de longitud variable, este cálculo no tiene lugar.

UTILIZACIÓN DE LA IMPRESORA

Para obtener resultados impresos Visual Basic provee el objeto **Printer**, el cual es utilizado para controlar el texto y los gráficos que se imprimen en una página y para enviarlos directamente a la impresora por defecto. Por ejemplo,

```
Printer.Print Libros.Título, Tab(40); Libros.Editorial
```

Para más detalles ver en el Capítulo 12 "IMPRESIÓN DE RESULTADOS".

FICHEROS INDEXADOS

Visual Basic no tiene ficheros indexados. No obstante, en el Capítulo 12 se estudiará su estructura, creación y mantenimiento.

CAPÍTULO 9

EFECTOS GRÁFICOS

INTRODUCCIÓN

Visual Basic provee varias formas de producir efectos gráficos y de una forma más sencilla que con la programación tradicional. Se pueden mover objetos por la pantalla, hacerlos aparecer y desaparecer, modificar su tamaño y además realizar cualquier gráfico que se nos pueda ocurrir, combinando las figuras básicas como líneas, cajas, círculos, etc., soportadas directamente por Visual Basic.

IMÁGENES

En Visual Basic es posible añadir una imagen (una figura en general) a una forma. La imagen puede ser puesta directamente sobre la forma o dentro de una caja de imagen (control imagen). Esta imagen puede estar almacenada en tres tipos de ficheros: .BMP (**bitmap**), .ICO (**icon**) y .WMF (**Windows metafile**).

Estos ficheros pueden ser creados por muchos paquetes gráficos. Por ejemplo, Windows puede crear ficheros .BMP por medio de la utilidad gráfica **Paintbrush**. Por otra parte, Visual Basic provee una amplia librería de ficheros .ICO (vea el directorio \vb\icons).

Cargar una imagen

Una imagen puede ser añadida durante el diseño de la aplicación y durante la ejecución.

Para añadir una imagen durante el diseño los pasos a seguir son:

1. Seleccione la forma o la caja de imagen.
2. Diríjase a la barra de propiedades, seleccione **Picture** de la lista de propiedades y haga clic en el botón [...] situado a la derecha de la barra de propiedades. Se visualiza una caja de diálogo.
3. Elija el fichero .BMP, .ICO, o .WMF que desee.

La imagen seleccionada es colocada en la forma o en la caja detrás de cualquier control que hubiera colocado allí.

También puede añadir una imagen de otra aplicación, utilizando el portapapeles. Por ejemplo, dibuje una imagen con PaintBrush, cópiela en el portapapeles y péguela en la forma o caja de imagen de su aplicación.

Para eliminar la imagen, ponga de nuevo la propiedad **Picture** a valor "(None)". Para ello, haga un doble clic en la barra de propiedades sobre el valor actual asignado a **Picture** y pulse la tecla **Del** (Supr).

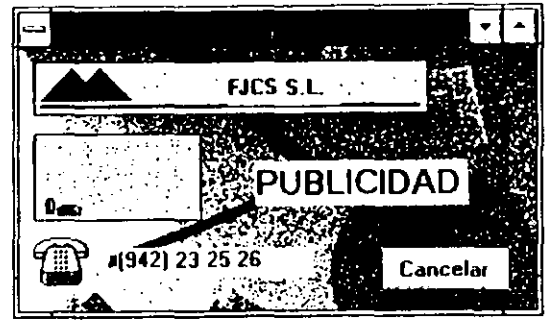
Para añadir una imagen durante la ejecución, utilice la función **LoadPicture**. Esta función permite asignar un fichero .BMP, .ICO o .WMF a la propiedad **Picture** de una forma o de una caja de imagen. Por ejemplo, diseñe una forma con los siguientes controles:

	Caption	CtlName	AutoSize	BorderStyle
Imagen 1		Imagen1	True	1
Etiqueta 1	FJCS S.L.	Etiqueta1		
Imagen 2		Imagen2	False	1
Imagen 3		Imagen3	True	0
Etiqueta 2	(942) 23 25 26	Etiqueta2		
Etiqueta 3	PUBLICIDAD	Etiqueta2		
Botón	Cancelar	Cancelar		

Escriba el procedimiento *Form_Load* para que utilizando la función **LoadPic** urgue en la forma *Form1* la imagen "fiesta.bmp", en *Imagen1* la imagen "bra p", en *Imagen2* la imagen "farm.wmf" y en *Imagen3* la imagen "phone01.ico". Ponga también en tiempo de ejecución los títulos de las etiquetas 1 y 2. El resultado se presenta a continuación.

La propiedad **AutoSize** con un valor -1 (**True**), permite que la caja aumente o disminuya automáticamente para ajustarse a la imagen. Si esta propiedad tiene valor 0 (**False**) y la imagen es más grande que el objeto que la va a contener, ésta es recortada por la derecha y por abajo. La propiedad **AutoSize** sólo es aplicable a etiquetas y cajas de imagen.

```
Sub Form_Load ()
  Form1.Picture = LoadPicture("c:\windows\fiesta.bmp")
  Etiqueta1.Caption = "FJCS S.L."
  Imagen1.Picture = LoadPicture("c:\vb\vb.cbt\brdrlk.bmp")
  Imagen2.Picture = LoadPicture("c:\vb\vb.cbt\farm.wmf")
  Imagen3.Picture = LoadPicture("c:\vb\icons\comm\phone01.ico")
  Etiqueta2.Caption = "(942) 23 25 26"
End Sub
```



También es posible asignar una imagen a un control, desde otro control. Si en el destino ya hay una imagen, ésta es sustituida por la nueva imagen. Por ejemplo,

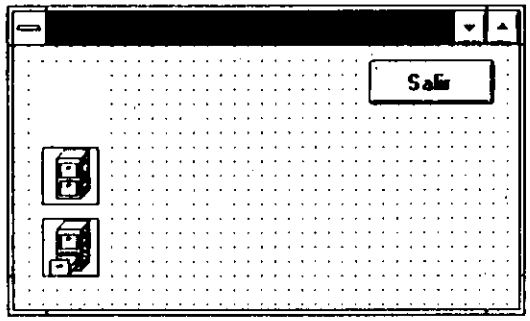
```
Imagen2.Picture = Imagen1.Picture
```

Las imágenes cargadas durante el diseño son guardadas y cargadas con la aplicación. Quiere esto decir que cuando cree un fichero .EXE de su aplicación, no necesitará acompañarlo de los ficheros de imágenes utilizados. En cambio, cuando las imágenes se cargan en tiempo de ejecución, utilizando la función **LoadPicture**, y cree un fichero .EXE de su aplicación, debe acompañar éste con los ficheros de imágenes utilizados.

En la librería de iconos de Visual Basic hay series de dos o más iconos que utilizándolos conjuntamente producen animación. Por ejemplo, fij los ico-

nos FILE03A.ICO (fichero cerrado) y FILE03B.ICO (fichero abierto). Para crear un efecto de animación utilizando estos iconos, siga los siguientes pasos:

1. Durante el diseño añada a la forma tres cajas de imagen.
2. Denomine a la primera *Fichero* y ponga su propiedad **BorderStyle** a 0.
3. Denomine a las otras dos, *FicheroCerrado* y *FicheroAbierto* y ponga la propiedad **Visible** de cada uno de ellos a valor 0. Significa esto que queremos utilizar las imágenes de estos controles, pero no queremos que se vean.
4. Asigne a la propiedad **Picture** del control *FicheroCerrado* la imagen del fichero cerrado (FILE03A.ICO) y a la propiedad **Picture** del control *FicheroAbierto*, la imagen del fichero abierto (FILE03B.ICO).



Recuerde ahora nuestra aplicación *Editor*. En el menú *Fichero* de esta aplicación había dos órdenes, *Abrir* y *Guardar*, que ahora queremos implementar de una forma simbólica, esto es, la orden *Abrir* queremos sustituirla por la imagen abrir el fichero (pasa a fichero abierto) y la orden *Guardar* por la imagen cerrar el fichero (pasa a fichero cerrado). El código en el que tiene que apoyarse para realizar este proceso se indica a continuación.

Primero, cuando la forma se cargue se inicializa la imagen *Fichero* para que muestre un fichero cerrado. Esta imagen aparecerá encima de las dos imágenes de la figura anterior, permaneciendo éstas ocultas.

```
Sub Form_Load ()
  Fichero.Picture = FicheroCerrado.Picture
End Sub
```

A continuación, cuando el usuario quiera abrir un fichero hará clic sobre la imagen *Fichero* para que la aplicación le permita seleccionar un fichero al mismo tiempo que se muestra la imagen fichero abierto. Cuando el usuario finalice la sesión y quiera guardar el fichero, hará clic sobre la imagen *Fichero* para que se realice ese proceso al mismo tiempo que se muestra la imagen fichero cerrado.


```
Sub Fichero_Click ()
  If Fichero.Picture = FicheroCerrado.Picture Then
    Fichero.Picture = FicheroAbierto.Picture
    'Mostrar la forma Sistema de Ficheros'
  Else
    Fichero.Picture = FicheroCerrado.Picture
    'Mostrar la forma Guardar Fichero
  End If
End Sub
```

OPERACIONES DINÁMICAS CON CONTROLES

Durante la ejecución de una aplicación se puede mover cualquier control a otra posición, modificar su tamaño, ocultarlo y añadir nuevos controles.

Para mover y/o para modificar el tamaño de un control durante la ejecución de una aplicación, lo único que hay que hacer es modificar sus propiedades **Left**, **Top**, **Width** y **Height** utilizando el método **Move**. La sintaxis para este método es la siguiente:

[*objeto.*]**Move** *izq*[, *sup*[, *ancho*[, *alto*]]]

Objeto se refiere a la forma o al control que se desea mover y/o redimensionar y los siguientes cuatro parámetros son datos reales de precisión simple cuyo significado es el siguiente: el parámetro *izq* especifica la distancia entre la esquina superior izquierda del control y el lado izquierdo de la forma; el parámetro *sup* especifica la distancia entre la esquina superior izquierda del control y el lado superior de la forma; el parámetro *ancho* especifica el ancho del control; y el parámetro *alto* especifica la altura del control. Por ejemplo,

```
Text1.Move 2500, 10
Text1.Move Text1.Left - 1000, Text1.Top - 300
```

La primera sentencia mueve el control *Text1* a las coordenadas (2500, 10); esta posición se mide respecto de la esquina superior izquierda de la forma. La segunda sentencia mueve el control *Text1* a la posición (1000, −300) relativa a la esquina superior izquierda del mismo.

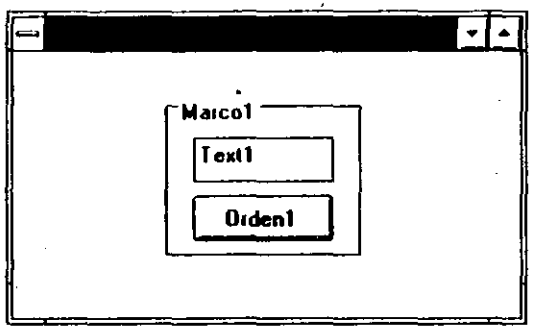
Todas las medidas empleadas para desplazar y dimensionar un objeto, así como las empleadas en las sentencias gráficas, son consideradas por Visual Basic

en twips. Recuerde que una pulgada son 1440 twips y por lo tanto, un centímetro 567 twips.

Como ejemplo diseñe una forma *Form1* con los siguientes controles:

	Caption	CtlName	Left	Top
Marco	Marco1	Marco1	1320	360
Caja de texto		Text1		
Botón	Orden1	Orden		

Recuerde, para agrupar controles bajo un marco, primero hay que diseñar el marco y después se incluyen los controles en el mismo. Para incluir un control, diríjase al panel de utilidades, haga un clic en el control deseado (no emplee la técnica de hacer un doble clic), apunte con el ratón al lugar dentro del marco donde desea dibujar el control, y con el botón izquierdo del ratón pulsado arrastre hasta dibujarlo. Por último, ajuste el tamaño del marco y de los controles.



Cuando el usuario pulse el botón *Orden1*, se tienen que realizar las siguientes operaciones:

- El marco se moverá a la coordenada (1000,-300) relativa a su posición y el alto se ampliará en 500 twips.
- El botón *Orden1* será sustituido por un botón *Restaurar*, que permitirá retornar el marco a su posición inicial con los parámetros y controles iniciales, y se añadirá el botón *Salir* que permitirá finalizar la aplicación (vea la figura de página 196).

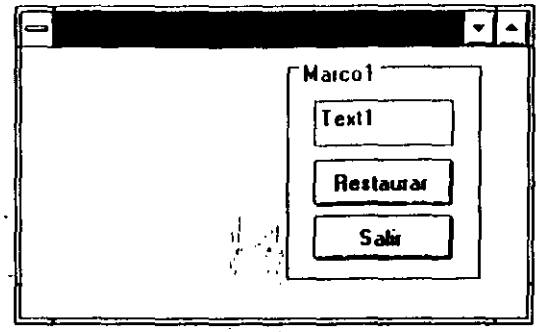
Para realizar el punto segundo, hay que añadir durante la ejecución dos nuevos controles: *Restaurar* y *Salir*, poner su propiedad **Visible** a valor -1 y situarlos en la posición adecuada. Como el control *Restaurar* queda encima de *Orden1*, hay que ocultar este último para que no intervenga.

Recuerde que para añadir un control durante la ejecución de una aplicación, este tiene que formar parte de un array de controles (vea el apartado "Creando controles en tiempo de ejecución" del Capítulo 5). Por lo tanto, en tiempo de diseño, ponga la propiedad **Index** del control denominado *Orden* a valor 0. Esto crea el array *Orden* con un elemento *Orden(0)* que referencia el botón titulado *Orden1*. A continuación escriba el procedimiento común *Orden_Click*.

```
Sub Orden_Click (Index As Integer)
  Select Case Index
    Case 0
      Marco1.Move Marco1.Left - 1000, Marco1.Top -
                  300, Marco1.Width, Marco1.Height + 500
      Load Orden(1)
      Load Orden(2)
      Orden(1).Caption = "Restaurar"
      Orden(1).Visible = -1
      Orden(2).Top = Orden(2).Top + 450
      Orden(2).Caption = "Salir"
      Orden(2).Visible = -1
      Orden(0).Visible = 0
    Case 1
      Marco1.Move Marco1.Left - 1000, Marco1.Top +
                  300, Marco1.Width, Marco1.Height - 500
      Unload Orden(1)
      Unload Orden(2)
      Orden(0).Visible = -1
    Case 2
      End
  End Select
End Sub
```

Cuando el usuario pulse el botón *Orden1* se ejecutará el procedimiento común asociado con el array *Orden* para el suceso **Click** (*Orden_Click*) para un valor de **Index** igual a 0. En este caso, **Move** mueve y redimensiona *Marco1*, **Load** carga los botones *Orden(1)* y *Orden(2)*, se asignan valores a las propiedades de estos nuevos controles y se oculta el botón denominado *Orden(0)*. En resumen, los botones titulados *Orden1*, *Restaurar* y *Salir* se corresponden con los nombres *Orden(0)*, *Orden(1)* y *Orden(2)* respectivamente.

Después de esto se visualiza la forma de la figura siguiente.



Cuando el usuario haga clic en el botón *Restaurar* se ejecutará el procedimiento *Orden_Click* para un valor de **Index** igual a 1. En este caso, **Move** retorna el marco a su posición y medidas iniciales, **Unload** descarga (elimina) los controles *Restaurar* y *Salir* y se hace visible el control *Orden1*.

Si el usuario hace clic en el botón *Salir*, la aplicación finaliza.

El método **Move** mueve y redimensiona el objeto en diagonal, esto es, simultáneamente en ambas direcciones lo que evita movimientos bruscos. Por ejemplo, la sentencia

```
Marco1.Move Marco1.Left - 1000, Marco1.Top -
          300, Marco1.Width, Marco1.Height + 500
```

puede descomponerse en las siguientes sentencias:

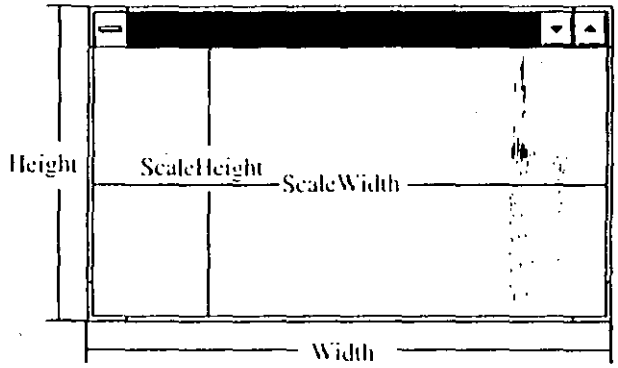
```
Marco1.Left = Marco1.Left + 1000
Marco1.Top = Marco1.Top - 300
Marco1.Height = Marco1.Height + 500
```

Pruebe esta segunda solución y observe como se producen saltos bruscos.

DIBUJANDO CON VISUAL BASIC

En Visual Basic se puede realizar un dibujo en una forma y en una caja de imagen. El origen de coordenadas para cualquiera de estos objetos coincide con la esquina superior izquierda de los mismos; la coordenada *x* se incrementa hacia la derecha y la coordenada *y* se incrementa hacia abajo.

Hay cuatro propiedades que especifican perfectamente las dimensiones de cada uno de estos objetos: **Width**, **Height**, **ScaleWidth** y **ScaleHeight** que se corresponden con el ancho y alto de la forma, y con el ancho y alto del área útil de la forma respectivamente.



Un sistema de coordenadas se puede ajustar a la escala que se desee. Hay siete escalas predefinidas, las cuales se indican a continuación.

Escala	Descripción
1	**Twips**. Hay 1440 twips por pulgada.
2	Puntos. Hay 72 puntos por pulgada.
3	**Pixels**. Un **pixel** es la unidad más pequeña de resolución de un monitor.
4	Caracteres. Un carácter se define como un área 1/6 pulgadas de alto y 1/12 pulgadas de ancho.
5	Pulgadas (1 pulgada es igual a 25,4 milímetros).
6	Milímetros.
7	Centímetros.

Para ajustar el sistema de coordenadas a una de estas escalas estándar hay que poner la propiedad **ScaleMode** al valor deseado. Por ejemplo,

```
ScaleMode = 6          ' Escala para la forma Form1
Imagen1.ScaleMode = 6 ' Escala caja de imagen Imagen1
```

Cuando se utiliza una escala estándar, excepto la 3, las unidades empleadas se corresponden con la longitud que se va a emplear para imprimir. Por ejemplo, utilizando la escala 6, un elemento de 5 unidades de longitud ocupará 5 milímetros cuando se escriba.

Para ajustar el sistema de coordenadas a la escala que nosotros deseemos, disponemos de las propiedades: **ScaleLeft**, **ScaleTop**, **ScaleWidth** y **ScaleHeight**. Por ejemplo, si queremos visualizar una curva *y=sen(x)* para valores de *x* de 0 a 6.3 sabiendo que los valores de *y* están comprendidos entre −1 y 1, un sistema de coordenadas adecuado es:

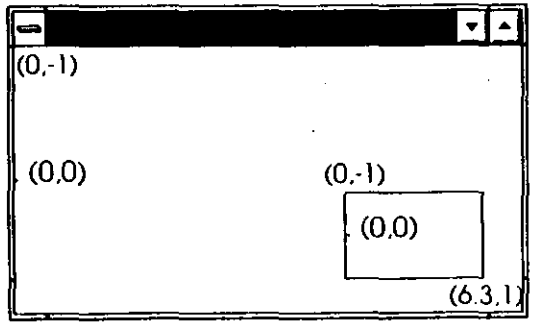
```
ScaleLeft = 0     'x de la esquina superior izquierda
ScaleTop = -1     'y de la esquina superior izquierda
ScaleWidth = 6.3  'ancho del área de dibujo de la forma
ScaleHeight = 2   'alto del área de dibujo de la forma
```

Las propiedades **ScaleLeft** y **ScaleTop** definen el valor numérico que se asigna a la esquina superior izquierda del objeto. Por lo tanto, el ejemplo anterior define un área de dibujo que comprende valores de *x* desde 0 hasta 6,3, ya que el ancho del área de dibujo es 6.3 unidades, y valores de *y* desde −1 a 1, ya que el alto del área de dibujo es 2 unidades. Por lo tanto, el origen de coordenadas queda situado a la izquierda y a la mitad del objeto como se muestra en la figura siguiente.

Cuando se utiliza **ScaleWidth** y **ScaleHeight** para definir un sistema de coordenadas, **ScaleMode** vale 0.

Esta definición de un sistema de coordenadas que acabamos de realizar, se puede hacer también utilizando el método **Scale**. Por ejemplo,

```
Scale (0, -1)-(6.3, 1)          'escala para la forma
Imagen1.Scale (0, -1)-(6.3, 1) 'escala para la caja
```



En general la sintaxis es,

[*objeto.*]**Scale** (*x1*, *y1*)-(*x2*, *y2*)

donde *x1* e *y1* se corresponden con **ScaleLeft** y **ScaleTop** y *x2* e *y2* con las sumas **ScaleWidth** + **ScaleLeft** y **ScaleHeight** + **ScaleTop** respectivamente.

Observe que para invertir el eje de coordenadas (valores positivos por encima del eje *x*) basta con cambiar de signo a **ScaleHeight** y asignar a **ScaleTop** el valor superior del eje *y*. Por ejemplo, para que los valores positivos queden por encima del eje *x* en la definición del sistema de coordenadas del ejemplo anterior, escriba:

```
ScaleLeft = 0     'x de la esquina superior izquierda
ScaleTop = 1      'y de la esquina superior izquierda
ScaleWidth = 6.3  'ancho del área de dibujo de la forma
ScaleHeight = -2  'alto del área de dibujo de la forma
```

o lo que es lo mismo,

```
Scale (0, 1)-(6.3, -1)          'escala para la forma
```

Escribir texto

Desde el punto de vista gráfico, Visual Basic puede escribir texto en una forma, en una caja de imagen y en la impresora. Al decir desde el punto de vista gráfico, nos referimos a texto que no aparece en controles orientados a texto, lo que significa que cuando escribimos un carácter en una forma o en una caja de imagen, no se almacena el carácter, sino que aparece una imagen de dicho carácter.

Para escribir texto en una forma, en una caja de imagen o en la impresora, utilizaremos el método **Print** cuya sintaxis es,

[*objeto.*]**Print** [*lista de expresiones*][{ ; | , }]

Si no se especifica el objeto sobre el cual queremos escribir, se supone la forma actual. Por ejemplo,

```
Print MsgS
Imagen1.Print MsgS
```

El texto escrito aparecerá a partir del origen de coordenadas. La razón es que la posición de salida para el texto viene fijada por las propiedades **CurrentX** y **CurrentY** que por defecto tienen valor 0. Quiere esto decir, que nosotros podemos fijar la posición de salida del texto, asignando las coordenadas de dicha posición a las propiedades **CurrentX** y **CurrentY**. Por ejemplo,

```
FontSize = 18
CX = ScaleWidth / 2 + ScaleLeft
CY = ScaleHeight / 2 + ScaleTop
Cls  ' limpiar la forma
MsgS = "Curvas"
' Coordenadas para escribir MsgS
CurrentX = CX - TextWidth(MsgS) / 2
CurrentY = CY + TextHeight(MsgS) / 2
' Color verde
ForeColor = QBColor(2)
Print MsgS
```

Este ejemplo fija para el texto un tamaño de 18 puntos, calcula el centro del área actual de dibujo (*CX*, *CY*), limpia la forma, calcula la posición de salida del texto para que quede centrado (**CurrentX**, **CurrentY**), fija el color verde para el primer plano y escribe el texto "Curvas".

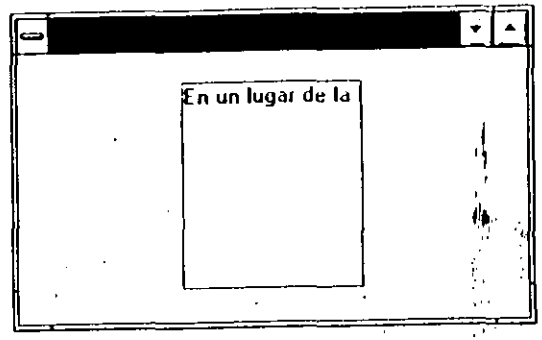
Los métodos **TextWidth** y **TextHeight** devuelven como resultado el ancho y el alto, respectivamente, del texto tal cual va a ser escrito en la forma, caja de imagen o impresora.

Si utiliza el procedimiento *Form_Load* para dibujar gráficos (líneas o texto) en una forma o en una caja de imagen, tiene que poner la propiedad **AutoRedraw** de ese objeto a valor **True** (−1), de lo contrario el gráfico no se visualizará.

Como ejemplo vamos a escribir texto en una caja de imagen. Inicie una nueva aplicación, ponga a la forma el título *Texto*, añada una caja de imagen, póngala el nombre *CajaTexto* y escriba el procedimiento que se indica a continuación. Guarde la aplicación y a continuación ejecútela.

```
Sub Form_Load ()
  Dim Texto As String
  Texto = "En un lugar de la Mancha de cuyo nombre "
  Texto = Texto + "no quiero acordarme ..."
  CajaTexto.Print Texto
End Sub
```

Observará que si no ha puesto la propiedad **AutoRedraw** a valor **True** no se visualiza nada. Si no lo ha hecho, hágalo ahora y ejecute de nuevo la aplicación. Ahora, como puede ver en la figura siguiente, el texto sí se visualiza pero la línea aparece cortada justamente en el ancho de la caja. La solución a este problema es formar a partir del texto, cadenas de caracteres que no superen el ancho de la caja de imagen y al mismo tiempo ir escribiendo una cadena a continuación de otra.



Para realizar el proceso descrito vamos a escribir un nuevo procedimiento denominado *FormatoTexto*. Este proceso será invocado desde el procedimiento *Form_Load* y recibirá como parámetros el objeto en el que vamos a escribir y el texto. Por lo tanto, el procedimiento *Form_Load* queda como sigue:

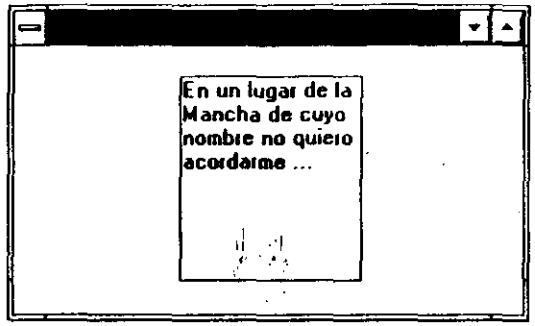
```
Sub Form_Load ()
  Dim Texto As String
  Texto = "En un lugar de la Mancha de cuyo nombre "
  Texto = Texto + "no quiero acordarme ..."
  Call FormatoTexto(CajaTexto, Texto)
End Sub
```

Abra la ventana de código para realizar un nuevo módulo (vea en el Capítulo 7 el apartado "Crear un módulo") cuyo esquema es el siguiente:

```
Sub FormatoTexto (Objeto As Control, Texto As String)

End Sub
```

En este esquema, el parámetro *Objeto* se refiere al área en la que vamos a escribir y el parámetro *Texto* al texto que aún nos falta por escribir.

Vamos a denominar *Línea* a la cadena de caracteres mayor (en nuestro ejemplo, la cadena es de palabras completas) que no supera el ancho de la caja y que por lo tanto dará lugar a la siguiente línea en la caja. La función *UnaPalabra* devuelve la siguiente palabra de *Texto* que añadiremos a *Línea* siempre y cuando la longitud actual de *Línea* más la de la palabra, no supere el ancho de la caja; si esto sucede, entonces escribimos *Línea* y la palabra devuelta por la función *UnaPalabra*, pasará a ser la primera de la siguiente *Línea*. Para saber si *Línea* más la siguiente palabra supera el ancho de la caja, utilizamos la variable temporal denominada *Temp*. Esta variable almacena *Línea* más la siguiente palabra que pretendemos añadir. El resultado que se obtiene se ve en la figura siguiente.



Con todo lo dicho el procedimiento *FormatoTexto* queda como se indica a continuación:

```
Sub FormatoTexto (Objeto As Control, Texto As String)
  Dim Línea As String, SiguientePalabra As String
  Dim Temp As String
  SiguientePalabra = UnaPalabra(Texto)
  Do
    Temp = Línea + SiguientePalabra
    If (Objeto.TextWidth(Temp) > Objeto.ScaleWidth) Then
      Objeto.Print Línea
      Línea = SiguientePalabra
    Else
      Línea = Temp
    End If
    SiguientePalabra = UnaPalabra(Texto)
  Loop Until (SiguientePalabra = "")
  Objeto.Print Línea
End Sub
```

La función *UnaPalabra* devuelve la primera palabra de la cadena *Texto* al mismo tiempo que elimina dicha palabra de esta cadena. De esta forma, cada vez que la función *UnaPalabra* es invocada, devuelve la siguiente palabra. Para tomar la siguiente palabra de *Texto* lo único que tenemos que hacer es buscar donde se encuentra el siguiente espacio en blanco. De esta tarea se encarga la función **InStr**(*XS*, *YS*) que devuelve la posición que ocupa el primer carácter de *YS* en *XS*. En nuestro caso la cadena *YS* es un espacio en blanco y *XS* es el texto que aún queda por escribir. Si *YS* no se encuentra en *XS* la función **InStr** devuelve un 0.

Visualice el módulo anterior y ejecute la orden **New Procedure** del menú **Code**, que le permitirá escribir en el mismo un nuevo procedimiento, en nuestro caso una nueva función, que realice lo anteriormente expuesto tal y como se especifica a continuación.


```
Function UnaPalabra (Texto As String) As String
  If (InStr(Texto, " ") = 0) Then
    UnaPalabra = Texto
    Texto = ""
  Else
    UnaPalabra = Left$(Texto, InStr(Texto, " "))
    Texto = Right$(Texto, Len(Texto) - InStr(Texto, " "))
  End If
End Function
```

Podrá completar el formato del texto utilizando las propiedades que se indican en la tabla siguiente:

Propiedad	Significado
FontName	Nombre de la fuente (por ejemplo, *Courier*)
FontSize	Tamaño en puntos de la fuente.
FontBold	Si su valor es **True** (-1) la fuente se visualiza en negra.
FontItalic	Si su valor es **True** (-1) la fuente se visualiza en cursiva.
FontStrikethru	Si su valor es **True** (-1) la fuente se visualiza tachada.
FontUnderline	Si su valor es **True** (-1) la fuente se visualiza subrayada.

Gráficos persistentes

Cuando la propiedad **AutoRedraw** de una forma o de una caja de imagen tiene un valor **True**, Visual Basic crea una imagen del objeto en memoria con el fin de poder reproducir automáticamente los gráficos (líneas o texto) cuando dicho objeto es maximizado, vuelto a visualizar en su forma habitual después de haberlo ocultado o minimizado, o cuando se agranda su tamaño. Si la propiedad **AutoRedraw** tiene un valor **False** (0), los gráficos no serán reproducidos.

Esta operación tiene un coste de memoria. En el caso de una forma con su propiedad **AutoRedraw** a valor **True** la imagen de memoria es de toda la pantalla y en caso de una caja de imagen sólo es de la caja. Por ello, si quiere conservar memoria tiene dos opciones: crear una caja de imagen para todos los dibujos y poner la propiedad **AutoRedraw** de la forma a valor **False**, o poner a valor **False** la propiedad **AutoRedraw** de la forma y de todas las cajas de imagen, y reproducir los gráficos directamente.

Para reproducir los gráficos directamente hay que incluir el código necesario en un procedimiento ligado al objeto (forma o caja de imagen) y controlado por el suceso **Paint** ya que éste se da cada vez que el objeto necesite ser [illegible]ado (por

ejemplo, cuando el objeto recupera su forma habitual después de haber sido minimizado). Este suceso no se da si la propiedad **AutoRedraw** está a valor **True**. La propiedad **AutoRedraw** puede ser modificada en tiempo de ejecución.

Limpiar el área de dibujo

El método **Cls** borra un gráfico (líneas o texto) generado en tiempo de ejecución en una forma o en una caja de imagen. Su sintaxis es,

[*objeto*].**Cls**

Los gráficos dibujados en un objeto mientras la propiedad **AutoRedraw** estaba a valor **True**, no se ven afectados por el método **Cls** mientras dicha propiedad esté a valor **False**. Las propiedades **CurrentX** y **CurrentY** son puestas a 0.

Dibujar puntos

Para dibujar un punto en la pantalla Visual Basic dispone del método **Pset**. Este método pone color al **pixel** de la posición especificada. La sintaxis para este método es la siguiente:

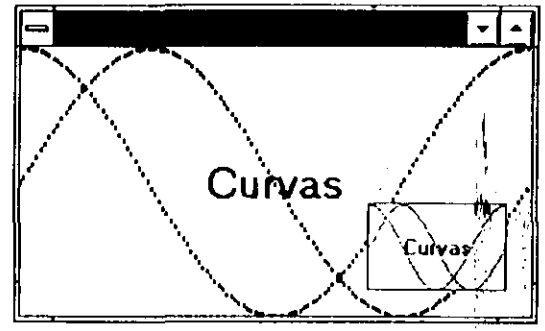
[*objeto.*]**Pset**(*X, Y*)[, *color*]

Los argumentos *X* e *Y* son valores reales de precisión simple que indican las coordenadas del punto y *color* es un argumento opcional que especifica el color que se quiere poner a dicho punto. Si *color* no se especifica se supone el color del primer plano (**ForeColor**). Las propiedades **CurrentX** y **CurrentY** son puestas, respectivamente, a los valores *X* e *Y*.

Como ejemplo, según se muestra en la figura siguiente, dibuje en una forma las curvas correspondientes a las funciones *sen(x)* y *cos(x)* para valores de *x* entre 0 y 6.3. Paralelamente realice el mismo gráfico en una caja de imagen.

Para empezar, cree una nueva aplicación y asigne a la forma el título *Gráficos*. A continuación añada, como se indica en la figura siguiente, una caja de imagen y póngala el nombre *Imagen1*. Ajuste el tamaño de la forma y de la caja y guarde la aplicación.

Para pintar las curvas, abra la ventana de código correspondiente a la forma y cree un procedimiento general denominado *Curvas* que haga lo siguiente:



- Crear un sistema de coordenadas para valores de *x* entre 0 y 6.3 y para valores de *y* entre −1 y 1, de forma que los valores positivos queden por encima del eje *x* lo que implica dar un valor negativo a la propiedad **ScaleHeight**.
- Fijar el grosor de la línea a 2, propiedad **DrawWidth**, y el tamaño de los caracteres gráficos a 18, propiedad **FontSize**.
- Escribir el título "Curvas" centrado en la forma y en la caja.
- Dibujar las curvas.

El procedimiento completo se especifica a continuación.

```
Sub Curvas ()
  ' Parámetros para la forma
  ScaleLeft = 0
  ScaleTop = 1
  ScaleWidth = 6.3
  ScaleHeight = -2
  DrawWidth = 2 ' grosor de la línea
  FontSize = 18 ' tamaño de los caracteres
  ' Coordenadas del punto central de la forma
  CX = ScaleWidth / 2 + ScaleLeft
  CY = ScaleHeight / 2 + ScaleTop
  Cls
  Msg$ = "Curvas"
  ' Coordenadas para escribir Msg$ centrado
  CurrentX = CX - TextWidth(Msg$)   2
  CurrentY = CY + TextHeight(Msg$)   2
  ' Color verde
  ForeColor = QBColor(2)
  Print Msg$
  ' Color negro
  ForeColor = QBColor(0)
  ' Parámetros para la imagen
  Imagen1.Scale (0, 1)-(6.3, -1
  Imagen1.Cls
```

```
    Imagen1.CurrentX = CX - Imagen1.TextWidth(Msg$) / 2
    Imagen1.CurrentY = CY + Imagen1.TextHeight(Msg$) / 2
    Imagen1.ForeColor = QBColor(2)
    Imagen1.Print Msg$
    Imagen1.ForeColor = QBColor(0)

    'Dibujar curvas
    For x = 0 To 6.3 Step .05
      yc = Cos(x): ys = Sin(x)
      PSet (x, yc): Imagen1.PSet (x, yc) ' coseno
      PSet (x, ys): Imagen1.PSet (x, ys) ' seno
    Next x
End Sub
```

El procedimiento *Curvas* será invocado desde el procedimiento *Form_Click* como se indica a continuación:

```
Sub Form_Click ()
  Curvas
End Sub
```

Ejecute la aplicación, haga clic sobre la forma y vea los resultados. Observe que como el procedimiento *Curvas* no es llamado desde el procedimiento *Form_Load*, los gráficos se visualizarán independientemente de que la propiedad **AutoRedraw**, de la forma y/o de la caja, esté a valor **False** o **True**. Pero hay una diferencia: si la propiedad **AutoRedraw** es **False** (0), los gráficos (líneas o texto) se escriben solamente en la pantalla, no en memoria, cosa que sí ocurre si la propiedad **AutoRedraw** es **True** (–1).

Suponiendo que en nuestro ejemplo la propiedad **AutoRedraw** es **False** (0), ejecute la aplicación y minimice la forma. Cuando la restaure, observará que los gráficos no persisten. Recuerde que Visual Basic restaura los gráficos solamente si la propiedad **AutoRedraw** es **True** (–1). No obstante, sabiendo que al agrandar una forma se da el suceso **Paint** puede hacer usted directamente la restauración de los gráficos desde el procedimiento *Form_Paint*. El proceso es sencillo: llame *Form_Paint* al procedimiento *Curvas* y modifique el procedimiento *Form_Click* como se indica a continuación:

```
Sub Form_Click ()
  Form_Paint ' Curvas
End Sub
```

Cada vez que el usuario haga clic sobre la forma, se ejecuta el procedimiento *Form_Click* y se pintan las curvas. Y también se ejecutará este procedimiento

cada vez que la forma necesite ser redibujada (por ejemplo, si se minimiza y después se restaura a su tamaño, o si se agranda su tamaño).

Dibujar líneas

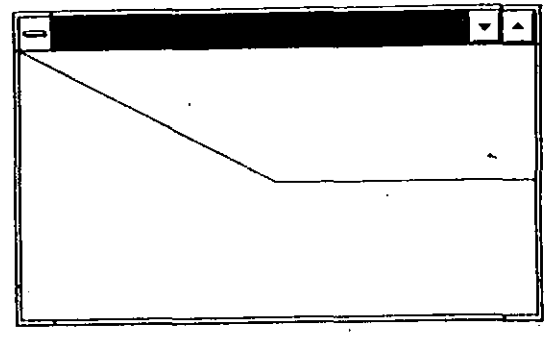
Para dibujar una línea entre dos puntos Visual Basic dispone del método **Line**, cuya sintaxis es,

[*objeto*.]**Line**[(*X1*, *Y1*)]–(*X2*, *Y2*)[, *color*]

Si *objeto* no se especifica se asume que la línea se dibuja sobre la forma actual. Las coordenadas pueden especificarse como valores enteros o reales. Si el punto (*X1*, *Y1*) que es opcional no se especifica, se supone la posición actual. Por ejemplo,

```
Sub Form_Click ()
  Line (0, 0)-(ScaleWidth / 2, ScaleHeight / 2)
  Line -(ScaleWidth, ScaleHeight / 2)
End Sub
```

Cuando el usuario haga clic sobre la forma se dibujarán las dos líneas que se muestran en la figura siguiente. Observe como la segunda línea empieza donde finalizó la primera.



Las propiedades **CurrentX** y **CurrentY** son automáticamente actualizadas al último punto de la línea.

Las coordenadas pueden expresarse también en forma relativa utilizando la palabra reservada **Step**. Por ejemplo,

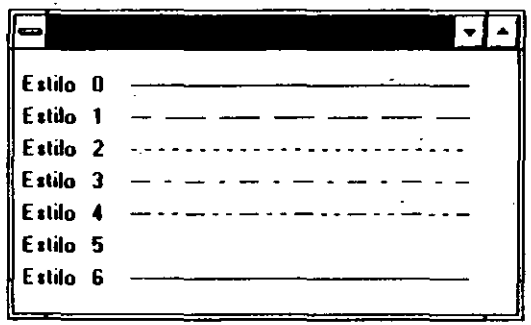
```
Line -Step(500, 500)
Line Step(200, 0)-Step(500, -100)
```

Estas sentencias son equivalentes a las dos siguientes:

```
Line (CurrentX, CurrentY)-(CurrentX + 311, CurrentY + 500)
Line (CurrentX + 200, CurrentY)-(CurrentX + 500, CurrentY - 100)
```

Si necesita variar el grosor de una línea utilice la propiedad **DrawWidth** y si necesita cambiar el estilo utilice la propiedad **DrawStyle**.

Como ejemplo, diseñe una forma y dibuje en ella los siete estilos de línea existentes (**DrawStyle** = 0 a **DrawStyle** = 6). El resultado que se quiere obtener se presenta en la siguiente figura. Observe que el estilo 5 es una línea invisible. El estilo 0 se denomina línea sólida y es el estilo por defecto; el estilo 6 se denomina línea sólida interior y es igual que el estilo 0, excepto cuando se utiliza para dibujar cajas como veremos a continuación.



El procedimiento conducido por el suceso **Click** que da lugar a los resultados pedidos es el siguiente:

```
Sub Form_Click ()
  Dim I As Integer, Y As Integer, H As Integer, Msg As String
  Msg = " Estilo "
  H = TextHeight(Msg) / 2
  For I = 0 To 6
    DrawStyle = I
    CurrentX = 0
    CurrentY = CurrentY - 175
    Print Msg; I;
    Line (1000, CurrentY - H)-(4110, CurrentY + H)
  Next I
End Sub
```

Un ancho de línea mayor que 1 da lugar a que los estilos 1 a 4 produzcan líneas sólidas.

Hasta ahora, nosotros hemos aprendido a cambiar el espesor de las líneas y su estilo. Para entender mejor esto, piense que cada objeto gráfico tiene una pluma que puede dibujar en diferentes estilos, dependiendo éstos de las propiedades que se definan (ancho, estilo, color). Dando un paso más, piense ahora como puede afectar lo que dibuja, si lo hace encima de un gráfico existente. Este concepto es el que define la propiedad **DrawMode**.

La propiedad **DrawMode** puede tomar valores de 1 a 16. Los valores más utilizados son los siguientes:

DrawMode	Nombre	Significado
4	Not pluma	Se dibuja el inverso de la pluma.
6	Inverso	Se dibuja el inverso de la pantalla.
7	Xor	Se dibuja el resultado de hacer la operación **Xor** entre la pluma y la pantalla.
10	Not Xor	Inversa de **Xor**.
11	No operación	No dibuja; la salida permanece igual.
13	Copiar pluma	Se dibuja directamente lo de la pluma. Es el valor que se toma por defecto.

Como veremos más adelante, el modo de dibujo 7 (**DrawMode** = 7) es útil para producir animación. Esto es posible porque dibujando en este modo un objeto dos veces, el plano de fondo queda restaurado a su valor inicial.

Dibujar cajas

Para dibujar una caja se utiliza el método **Line** que acabamos de exponer, pero añadiendo la opción **B** según se especifica a continuación.

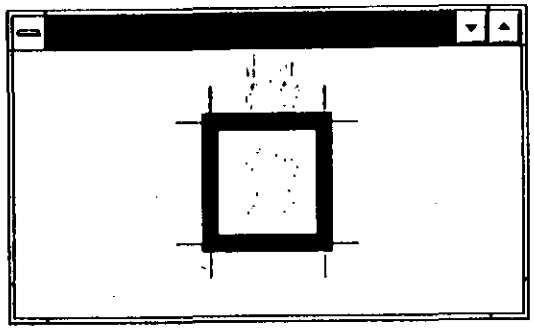
[*objeto.*]**Line** [(*X1!*, *Y1!*)] – (*X2!*, *Y2!*) [,[*color&*],**B**[**F**]]

Cuando se especifica la opción **B**, las coordenadas (*X1*, *Y1*) y (*X2*, *Y2*) hacen referencia a las esquinas superior izquierda e inferior derecha de la caja. Si se especifica la opción **BF** se obtiene una caja coloreada. Por ejemplo,

```
Line (100, 100)-(1000, 1000), 255, BF
```

Anteriormente dijimos que a la hora de dibujar cajas, había una diferencia entre los estilos 0 y 6. Esta diferencia, queda definida con las dos figuras siguientes. Las medidas de la caja en esas dos figuras, se corresponden con lo indicado

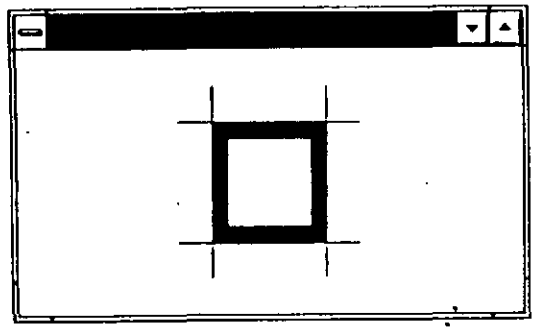
por las líneas finas. Esa misma caja se ha dibujado con un ancho de línea de 10 y estilo 1; se observa como la línea gruesa que dibuja la caja se reparte por igual a ambos lados de la línea fina. En cambio si definimos el estilo 6, la línea gruesa que dibuja la caja es interior a las líneas finas.



El procedimiento que da lugar a la figura anterior es,

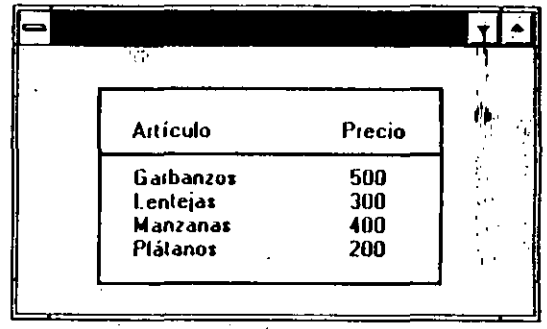
```
Sub Form_Click ()
  Line (1400, 600)-(3000, 600)
  Line (1400, 1600)-(3000, 1600)
  Line (1700, 300)-(1700, 1900)
  Line (2700, 300)-(2700, 1900)
  DrawWidth = 10
  DrawStyle = 1
  Line (1700, 600)-(2700, 1600), , B
End Sub
```

Ahora, cambie el estilo a 6 poniendo `DrawStyle = 6`. El resultado es la figura siguiente:



(ejemplo, diseñe una caja como la de la figura siguiente. Para ello, co-
mience a nueva aplicación, ponga a la forma el título *Cajas* y a continuación

abra la ventana de código y cree el procedimiento *Form_Load* para que escriba las líneas de texto especificadas y dibuje la caja que las contiene.



El código para el procedimiento que da lugar al gráfico anterior, se presenta a continuación.

```
Sub Form_Load ()
  AutoRedraw = -1
  Print
  Print
  Print
  Print Tab(12); "Artículo"; Tab(32); "Precio"
  Print
  Print Tab(12); "Garbanzos"; Tab(32); 500
  Print Tab(12); "Lentejas"; Tab(32); 300
  Print Tab(12); "Manzanas"; Tab(32); 400
  Print Tab(12); "Plátanos"; Tab(32); 200
  DrawWidth = 2
  Line (700, 350)-(3650, 1950), , B
  Line (700, 880)-(3650, 880)
End Sub
```

La función **Tab** indica la posición (columna para un carácter) donde se desea que comience la impresión de una determinada expresión.

Dibujar círculos, elipses y arcos

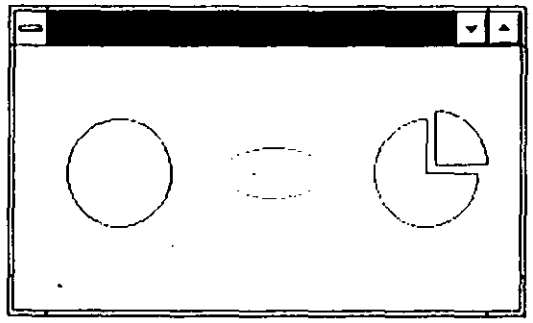
Para dibujar un círculo, una elipse o parte de estas figuras con centro (*X, Y*) en la pantalla utilizaremos el método Circle. La sintaxis para este método es,

[*objeto.*]**Circle** (*X!, Y!*), *radio!*[, [*color&*][, [*comienzo!*][, [*final!*][, *as*[illegible]*!*]]]]

El argumento *radio* es el radio del círculo o el eje mayor de la elipse. Ambos se miden en sentido horizontal si *aspecto* < *1* y en sentido vertical si *aspecto* > *1*. El argumento *aspecto* es una expresión numérica que afecta a la relación X/Y.

Los argumentos *comienzo* y *final* son ángulos en radianes, de valores comprendidos en el rango -2π a 2π, que especifican donde comienza y finaliza el arco. Si se omiten, se traza una curva completa. Si se les antepone un signo menos, se dibujarán los radios correspondientes desde las terminaciones del arco al centro.

Como ejemplo, diseñe una nueva aplicación, ponga a la forma el título *Círculos* y a continuación abra la ventana de código y cree el procedimiento *Form_Load* para que dibuje un círculo, una elipse y dos sectores circulares complementarios como se muestra en la figura siguiente.



El código para el procedimiento que da lugar al gráfico anterior, se presenta a continuación.

```
Sub Form_Load ()
  Dim Pi As Double
  Dim C1 As Long, C2 As Long, C3 As Long
  Pi = 3.1415926
  C1 = RGB(0, 0, 255): C2 = RGB(0, 255, 0): C3 = RGB(255, 0, 0)
  AutoRedraw = -1
  ScaleMode = 3
  Circle (60, 70), 30, C1 'círculo
  Circle (150, 70), 30, C2, , , 5 / 11 'elipse
  Circle (240, 70), 30, C3, -Pi / 2, -2 * Pi 'sector
  Circle Step(5, -5), 30, C1, -2 * Pi, -Pi / 2 'sector
End Sub
```

Observe que se ha empleado la escala estándar 3, que significa que las coordenadas son tomadas en **pixels**.

Las propiedades **CurrentX** y **CurrentY** son puestas, respectivamente, a los valores *X* e *Y* especificados en **Circle**.

Colorear figuras

Una figura puede ser rellenada con un color sólido o con un determinado patrón. Para seleccionar con qué se va a rellenar la figura, utilizaremos la propiedad **FillStyle** de la misma. Los valores posibles para esta propiedad se indican en una tabla a continuación.

Valor para FillStyle	Significado
0	Color sólido
1	Color transparente. Es la opción por defecto.
2	Líneas horizontales
3	Líneas verticales
4	Diagonales ascendentes
5	Diagonales descendentes
6	Líneas cruzadas
7	Diagonales cruzadas

Si para rellenar una figura se utiliza un color sólido (**FillStyle** = 0), por defecto se utiliza el color negro. Para especificar otro color, utilice la propiedad **FillColor**. El valor para esta propiedad puede ser especificado por una de las funciones **RGB** o **QBColor**, o también pueden especificarse algunos de los colores predefinidos en el fichero CONSTANT.TXT. Por ejemplo,

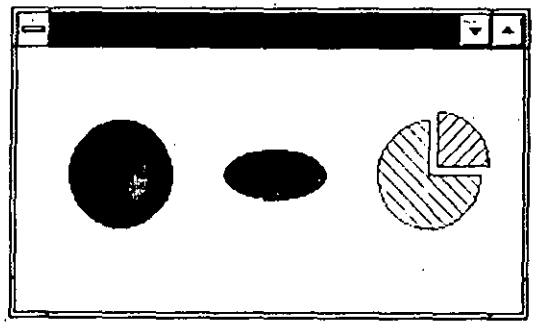
```
FillColor = RGB(0, 255, 0)
Form1.FillColor = QBColor(2)
Imagen1.FillColor = GREEN ' definido en CONSTANT.TXT
```

Como ejemplo coloree el círculo y la elipse de la figura anterior con un color sólido y los sectores circulares con algunos de los patrones. Una posible solución puede ser la que se muestra a continuación.

```
Sub Form_Load ()
  Dim Pi As Double
  Dim C1 As Long, C2 As Long, C3 As Long
  Pi = 3.1415926
  C1 = RGB(0, 0, 255): C2 = RGB(0, 255, 0): C3 = RGB(255, 0, 0)
  AutoRedraw = -1
  ScaleMode = 3
```

```
    FillStyle = 0
    FillColor = C1
    Circle (60, 70), 30, C1 'círculo
    FillColor = C2
    Circle (150, 70), 30, C2, , , 5 / 11 'elipse
    FillColor = C3: FillStyle = 4
    Circle (240, 70), 30, C3, -Pi / 2, -2 * Pi 'sector
    FillColor = C1: FillStyle = 5
    Circle Step(5, -5), 30, C1, -2 * Pi, -Pi / 2 'sector
End Sub
```

El resultado que se obtiene se muestra a continuación.



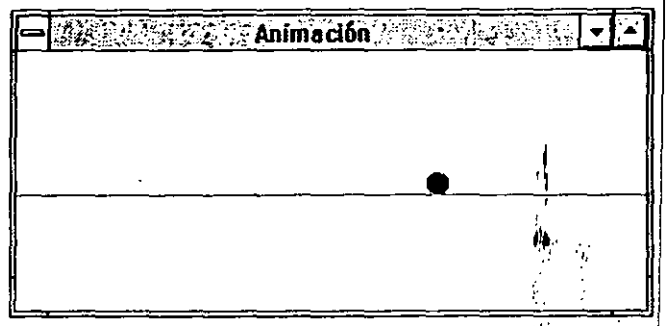
Animación de gráficos

La animación de una imagen se realiza de acuerdo con la siguiente secuencia de pasos:

1. Se dibuja una imagen en la pantalla.
2. Se introduce un retardo que será función de la velocidad con la que se quiere que se mueva el objeto.
3. Se borra la imagen, se calcula una nueva posición y se repite la secuencia de pasos desde el 1.

Como ejemplo, vamos a diseñar una aplicación que muestre una pelota rodando a lo largo de una línea, según se muestra en la figura siguiente.

Para ello, diseñe una nueva aplicación, ponga a la forma el título *Animación* y a continuación abra la ventana de código y cree el procedimiento *Form_Load* como se muestra a continuación.

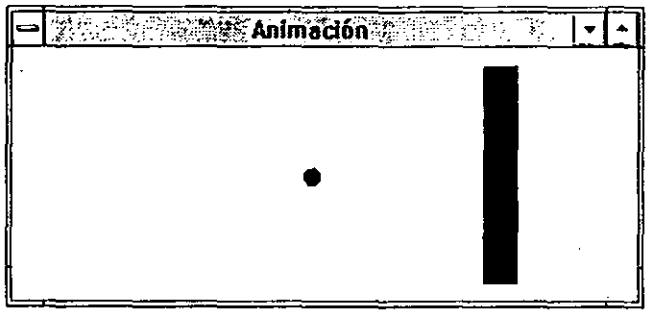



```
Sub Form_Click ()
  Dim I As Integer, Retar As Integer, D As Integer
  D = 100 'radio de la pelota
  BackColor = RGB(255, 255, 255) 'blanco
  ForeColor = RGB(0, 0, 255)     'azul
  CY = ScaleHeight / 2

  'Línea sobre la que rueda la pelota
  Line (0, CY + D)-(ScaleWidth, CY + D), QBColor(2)

  FillStyle = 0 'color sólido
  FillColor = ForeColor 'color pelota
  DrawMode = 10 'Not Xor

  'Animación - pelota rodando
  For I = 0 To ScaleWidth Step 30
    Circle (I, CY), D 'dibujar pelota
    For Retar = 0 To 1000: Next 'control velocidad
    Circle (I, CY), D 'borrar pelota
  Next I
End Sub
```

En este procedimiento, lo primero que se hace es fijar el color de fondo y del primer plano. A continuación se dibuja la línea sobre la que va a correr la pelota. Las sentencias **FillStyle = 0** y **FillColor = ForeColor** permiten rellenar la pelota del color especificado cada vez que se dibuja. Por último, se activa el modo de dibujo 10 (**Xor**, negada, entre la pluma y la pantalla, porque el fondo es blanco) y se efectúa la animación de la figura. Observe que primero se dibuja la pelota en una posición determinada, después se produce un retardo (**For** *Retar*) y se dibuja la pelota por segunda vez en la misma posición. Esto hace que la pelota se borre, ya que cada vez que se dibuja, se está haciendo la operación **Xor**, negada, entre la pluma y la pantalla. El siguiente paso es dibujar la pelota en la siguiente posición y así sucesivamente.

El efecto exacto de **DrawMode** depende del color de la figura que se dibuja y de los colores de la pantalla. Por ejemplo, el modo de dibujo 7 realiza la operación **Xor** de cada **pixel** que se dibuja con cada **pixel** de la pantalla. Si el punto de la pantalla es negro (todo ceros) la operación **Xor** es el propio punto dibujado; ahora si el punto de la pantalla es blanco (todo unos) la operación **Xor** es el punto dibujado, pero negado; en cambio, si en este último caso la operación es **Not Xor**, es resultado es el propio punto.

El siguiente ejemplo presenta una pelota que rebota al chocar contra una barrera.



Diseñe una nueva aplicación, ponga a la forma el título *Animación* y a continuación abra la ventana de código y cree el procedimiento *Form_Load* como se muestra a continuación.

```
Sub Form_Click ()
  Dim I As Integer, Retar As Integer
  Dim D As Integer, Dirección As Integer
  ScaleMode = 3 'escala en pixels
  D = 5 'radio de la pelota
  BackColor = QBColor(15) 'blanco
  ForeColor = QBColor(1)  'azul
  CX = ScaleWidth / 4 * 3
  CY = ScaleHeight / 2

  'Dibujar la barrera sobre la que rebota la pelota
  Line (CX, 10)-(CX + 20, ScaleHeight - 10), QBColor(2), BF

  FillStyle = 0 'color sólido
  FillColor = ForeColor 'color pelota
  DrawMode = 10 'Not Xor
  '[illegible] la pelota por la pantalla.
  '[illegible] con la barrera rebota.
  Dirección = 1 '1 = derecha, -1 = izquierda
  [illegible]
```



```
  Do While (I > 3 Or Dirección = 1)
    Circle (I, CY), D 'dibuja pelota
    'Point da el color de la barrera cuando hay contacto
    If Point(I + 6, CY) <> BackColor Then Dirección = -1
    For Retar = 0 To 300: Next 'control velocidad
    Circle (I, CY), D 'borra pelota
    I = I + Dirección
  Loop
End Sub
```

Observe que mientras la variable *Dirección* valga 1, la pelota se mueve hacia la derecha y que cuando vale -1 la pelota se mueve hacia la izquierda. Esto es así, porque la coordenada X (variable *I*) cuando se dibuja la pelota se ve incrementada o decrementada en el valor de *Dirección*.

El método **Point** da como resultado el color RGB de un punto (*X*, *Y*), correspondiente a una forma o a una caja de imagen. Su sintaxis es,

[*objeto*.]**Point**(*X*, *Y*)

Observe que para saber si la pelota ha llegado a tocar la barrera, cada vez que se dibuja se analiza el color del punto justamente a continuación de la misma. Si el color coincide con el color de fondo, quiere decir que aún no se ha llegado a la barrera.

CAPÍTULO 10

SUCESOS DEL RATÓN

INTRODUCCIÓN

Las formas y varios tipos de controles reconocen tres sucesos producidos por el ratón:

- **MouseDown**. Reconocido cuando el usuario pulsa cualquier botón.
- **MouseUp**. Reconocido cuando el usuario suelta cualquier botón.
- **MouseMove**. Reconocido cada vez que el usuario mueve el puntero del ratón a una nueva posición.

Una forma reconoce un suceso del ratón cuando el puntero del mismo está en una zona en la que no hay ningún control; y un control reconoce un suceso del ratón cuando el puntero del ratón está sobre el propio control.

Cuando el usuario pulsa un botón del ratón sobre un objeto y lo mantiene pulsado, después que el objeto haya reconocido el suceso **MouseDown**, aunque mueva el puntero fuera del objeto, éste continúa reconociendo el suceso **MouseMove**, mientras se mueva el ratón, y el suceso **MouseUp** cuando suelte el botón del ratón.

ARGUMENTOS DE LOS SUCESOS DEL RATÓN

Los tres sucesos tienen los mismos argumentos: *Button*: botón que da lugar al suceso; *Shift*: tecla pulsada (**Shift**, **Ctrl** o **Alt**) o teclas pulsadas; *X*, *Y*: coordenadas correspondientes a la posición del puntero del ratón.

Argumento Button

Es un entero cuyos tres bits menos significativos indican el botón que da lugar al suceso. Por lo tanto, es un valor entre 0 y 7 dependiente del suceso y de los botones que se hayan pulsado simultáneamente. En la siguiente figura, I es el bit correspondiente al botón izquierdo. D es el bit correspondiente al botón derecho y M es el bit correspondiente al botón del medio. El resto de los bits no se utilizan.

0	0	0	0	0	0	0	0	0	0	0	0	0	M	D	I

Sucesos **MouseDown** y **MouseUp**:

Se ha pulsado el botón izquierdo (**Button** = 1)	0	0	1
Se ha pulsado el botón derecho (**Button** = 2)	0	1	0
Se ha pulsado el botón del medio (**Button** = 4)	1	0	0

Para los sucesos **MouseDown** y **MouseUp** solamente un bit del argumento *Button* se pone a 1, el correspondiente al botón que se ha pulsado. Esto quiere decir que es posible saber qué botón fue utilizado, pero no cuántos se pulsaron simultáneamente.

Para saber que botón fue utilizado y cuántos botones se pulsaron, hay que verificar el valor del argumento *Button* para el suceso **MouseMove**. Este valor puede ser, además de los anteriores, los de la tabla siguiente.

Sucesos **MouseMove**:

No hay botones pulsados (**Button** = 0)	0	0	0
Se pulsaron los botones izquierdo y derecho (**Button** = 3)	0	1	1
Se pulsaron los tres botones (**Button** = 7)	1	1	1

Para determinar qué botón causa el suceso **MouseDown** o **MouseUp**, escriba los siguientes procedimientos:


```
Sub Form_MouseDown (Button As Integer, Shift As
                    Integer, X As Single, Y As Single)
  Print "Se ha pulsado el ";
  If Button = 1 Then Print "botón izquierdo"
  If Button = 2 Then Print "botón derecho"
  If Button = 4 Then Print "botón del medio"
End Sub

Sub Form_MouseUp (Button As Integer, Shift As Integer,
                  X As Single, Y As Single)
  Print "Se ha soltado el ";
  If Button = 1 Then Print "botón izquierdo"
  If Button = 2 Then Print "botón derecho"
  If Button = 4 Then Print "botón del medio"
End Sub
```

Para determinar si se ha pulsado un solo botón o más de uno hay que recurrir al suceso **MouseMove**. El siguiente procedimiento presenta las combinaciones posibles.

```
Sub Form_MouseMove (Button As Integer, Shift As
                    Integer, X As Single, Y As Single)
  If Button = 0 Then Exit Sub
  If Button = 1 Or Button = 2 Or Button = 4 Then
    Print "Se pulsó sólo el botón ";
    If (Button = 1) Then Print "izdo"
    If (Button = 2) Then Print "dcho"
    If (Button = 4) Then Print "medio"
  Else
    Print "Se pulsaron los botones "
    If (Button And 1) Then Print "izdo y ?"
    If (Button And 2) Then Print "dcho y ?"
    If (Button And 4) Then Print "medio y ?"
    If (Button And 3) = 3 Then Print "izdo y dcho"
    If (Button And 7) = 7 Then Print "izdo, dcho y medio"
  End If
End Sub
```

Para saber si sólo se pulsó un botón (por ejemplo el izquierdo) la sentencia tiene que ser similar a la siguiente:

```
If (Button = 1) Then ...
```

Para saber si se pulsó un determinado botón (por ejemplo el izquierdo) independientemente de que se hayan o no pulsado otros, la sentencia tiene que ser similar a la siguiente:

```
If (Button And 1) Then ...
```

Para saber si se pulsaron los botones izquierdo y derecho, la sentencia tiene que ser la siguiente:

```
If (Button And 3) = 3 Then ...
```

Observe que sentencias como las siguientes dan lugar a resultados erróneos.

```
If (Button And 3) Then ...
If (Button And 1) And (Button And 3) Then ...
```

Argumento Shift

La utilización del ratón puede ser más diversa cuando se utiliza conjuntamente con las teclas **Shift**, **Ctrl** o **Alt**.

El argumento *Shift* es un entero cuyos tres bits menos significativos indican las teclas pulsadas. Por lo tanto, es un valor entre 0 y 7 dependiente de las teclas que se hayan pulsado simultáneamente. En la siguiente figura, **S** es el bit correspondiente a la tecla **Shift**, **C** es el bit correspondiente a la tecla **Ctrl** y **A** es el bit correspondiente a la tecla **Alt**. El resto de los bits no se utilizan.

0	0	0	0	0	0	0	0	0	0	0	0	0	A	C	S

No hay teclas pulsadas (argumento *Shift* = 0)	0	0	0
Se ha pulsado la tecla **Shift** (*Shift* = 1)	0	0	1
Se ha pulsado la tecla **Ctrl** (*Shift* = 2)	0	1	0
Se ha pulsado la tecla **Alt** (*Shift* = 4)	1	0	0
Se han pulsado las teclas **Ctrl+Shift** (*Shift* = 3)	0	1	1
Se han pulsado las teclas **Alt+Shift** (*Shift* = 5)	1	0	1
Se han pulsado las teclas **Alt+Ctrl** (*Shift* = 6)	1	1	0
Se han pulsado las teclas **Alt+Ctrl+Shift** (*Shift* = 7)	1	1	1

Por ejemplo, para saber si conjuntamente con el botón izquierdo del ratón se ha pulsado la tecla **Alt**, la sentencia adecuada es:

```
If Button = 1 And Shift = 4 Then ...
```

El estudio para saber si se ha pulsado una o más teclas, es similar al realizado para los botones.

Argumentos X, Y

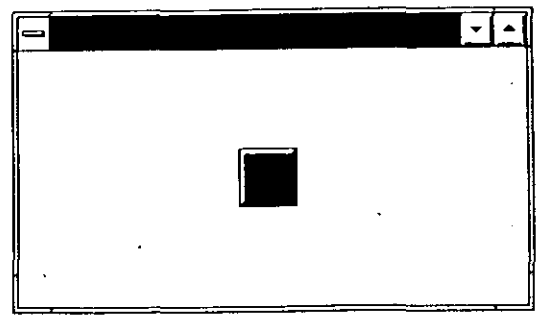
Los argumentos **X** e **Y** se corresponden con las coordenadas del puntero del ratón referidas al objeto para el que se da el suceso.

Como ejemplo cree una nueva aplicación llamada *raton.mak*. Cree una nueva forma llamada *raton.frm* y añadá a la misma los controles de la tabla siguiente:

Objeto	Propiedad	Valor
Caja de imagen	AutoSize Picture	True CIR_OF.ICO

El icono CIR_OF.ICO no se encuentra en las librerías de Visual Basic. Para crearlo utilice la aplicación ejemplo *iconwrks.mak* suministrada con Visual Basic.

La forma sería similar a la de la figura siguiente.



Nuestro objetivo es que cuando el usuario haga clic en una posición determinada de la forma, el objeto se mueva a esa posición. La caja de imagen tiene el nombre *Picture1* por defecto. Un procedimiento sencillo que realiza esta operación es el siguiente:

```
Sub Form_MouseDown (Button As Integer, Shift As
                    Integer, X As Single, Y As Single)
  Picture1.Move X, Y
End Sub
```

Observe que al desplazarse el objeto, se coloca su esquina superior izquierda en la posición indicada por el ratón. El método **Move** fue expuesto en el capítulo anterior.

A continuación haga que el objeto persiga al puntero del ratón de forma que su centro se coloque en la posición indicada por el ratón. El procedimiento que realice este proceso tiene que ejecutarse cuando se mueva el ratón.

```
Sub Form_MouseMove (Button As Integer, Shift As
                    Integer, X As Single, Y As Single)
  CX = X - Picture1.Width / 2
  CY = Y - Picture1.Height / 2
  Picture1.Move CX, CY
End Sub
```

Observe que un movimiento rápido del ratón genera saltos más grandes que si lo mueve más lentamente. Esto quiere decir que Visual Basic no genera necesariamente un suceso **MouseMove** por cada unidad de medida, sino que genera un número limitado de sucesos por segundo.

APLICACIONES GRÁFICAS

Vamos a realizar una aplicación simple que permita al usuario realizar dibujos sobre una forma, según se muestra en la siguiente figura. La aplicación debe cumplir las siguientes características:

- Mostrará en todo momento las coordenadas (X, Y) del puntero del ratón.

```
Form1.X.Caption = Str$(X)
Form1.Y.Caption = Str$(Y)
```

- Cuando el usuario apunte a una determinada posición y pulse el botón izquierdo del ratón, las coordenadas (X, Y) correspondientes a la posición ele-

se guardan en (*CX*, *CY*). También (X, Y) pasarán a ser las coordenadas actuales (**CurrentX**, **CurrentY**). Recuerde que el método **Line** modifica las propiedades **CurrentX** y **CurrentY**.

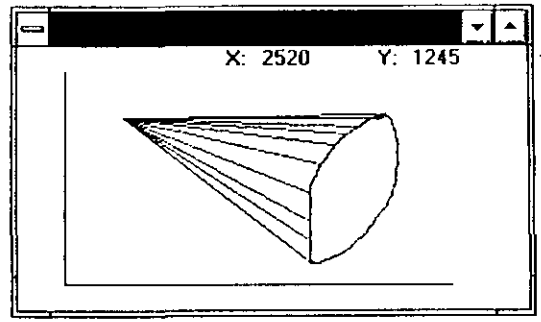
```
CX = X: CurrentX = X
CY = Y: CurrentY = Y
```

- Cuando el usuario arrastre el ratón manteniendo pulsado el botón izquierdo, realizará un dibujo continuo.

```
If Button = 1 Then
  Line -(X, Y)
End If
```

- Cuando el usuario pulse el botón derecho del ratón se dibujará una recta desde las coordenadas (*CX*, *CY*) a la posición señalada.

```
If Button = 2 Then
  Line (CX, CY)-(X, Y)
End If
```



Para realizar lo anteriormente expuesto, cree una nueva aplicación llamada *graficos.mak*. Cree una nueva forma llamada *graficos.frm*, póngala el título *Gráficos* y añada a la misma los controles de la tabla siguiente:

Objeto	Propiedad	Valor
Etiqueta 1	Caption	X
	CtlName	Label1
Etiqueta 2	Caption	(nada)
	CtlName	X
Etiqueta 3	Caption	Y
	CtlName	Label3
Etiqueta 4	Caption	(nada)
	CtlName	Y

Cuando el usuario pulse un botón del ratón será para fijar una coordenada de partida (botón izquierdo) o para dibujar una línea (botón derecho). Por lo tanto el suceso a tener en cuenta es *MouseDown*.

```
Sub Form_MouseDown (Button As Integer, Shift As
                    Integer, X As Single, Y As Single)
  If Button = 1 Then
    CX = X: CurrentX = X
    CY = Y: CurrentY = Y
  ElseIf Button = 2 Then
    Line (CX, CY)-(X, Y)
  End If
End Sub
```

Cuando el usuario quiera dibujar de un modo continuo, moverá el ratón en la dirección deseada con el botón izquierdo pulsado. Por lo tanto el suceso a tener en cuenta es *MouseMove*. También, siempre que se mueva el ratón hay que actualizar las coordenadas (*X*, *Y*) visualizadas en la forma.

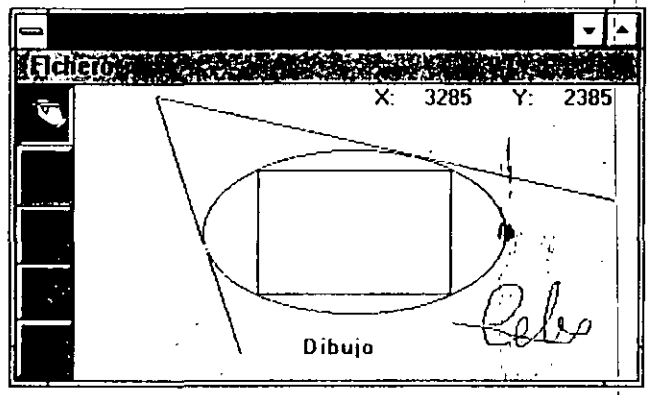
```
Sub Form_MouseMove (Button As Integer, Shift As
                    Integer, X As Single, Y As Single)
  Form1.X.Caption = Str$(X)
  Form1.Y.Caption = Str$(Y)
  If Button = 1 Then
    Line -(X, Y)
  End If
End Sub
```

EJEMPLO DE UN PANEL DE DIBUJO

La figura siguiente, presenta el aspecto final de la aplicación que tratamos de desarrollar. Como puede ver consta de un menú, un panel con cinco utilidades y un registro *X*, *Y* que presenta las coordenadas correspondientes a la posición actual del cursor del ratón.

El menú *Fichero* consta de dos órdenes: *Nuevo* y *Salir*. La orden *Nuevo* limpia la pantalla para iniciar un nuevo dibujo y la orden *Salir* finaliza una sesión de trabajo.

El panel de utilidades consta de las herramientas: dibujar a mano alzada, dibujar líneas, dibujar cajas, dibujar elipses o círculos y escribir texto. El cursor del ratón será una cruz sobre el área de dibujo y una flecha sobre los botones cuando se va a hacer clic sobre uno de ellos.



Para *dibujar a mano alzada*, primero haga clic sobre este botón. A continuación sitúe el cursor del ratón en la posición del panel donde desea comenzar a dibujar y pulse el botón izquierdo del mismo; con este botón pulsado arrastre en la dirección deseada. Para finalizar suelte el botón.

Para *dibujar líneas*, primero haga clic sobre este botón. A continuación sitúe el cursor del ratón en la posición del panel donde se desea que comience la línea, pulse el botón izquierdo del mismo y con él pulsado mueva el ratón en la dirección deseada hasta dejar la línea con la longitud e inclinación requeridas. Para finalizar suelte el botón.

Para *dibujar cajas*, primero haga clic sobre este botón. A continuación sitúe el cursor del ratón en la posición del panel donde desea situar la esquina superior izquierda de la caja, pulse el botón izquierdo del mismo y con él pulsado mueva el ratón en la dirección deseada hasta dibujar la caja. Para dibujar un cuadrado perfecto, pulse al mismo tiempo la tecla **Ctrl**. Para finalizar suelte el botón.

Para *dibujar círculos o elipses*, primero haga clic sobre este botón. A continuación sitúe el cursor del ratón en la posición del panel donde desea situar el centro de la elipse o del círculo, pulse el botón izquierdo del mismo y con él pulsado mueva el ratón en la dirección deseada hasta dibujar la elipse. Si quiere asegurarse de que lo que dibuja es un círculo, pulse al mismo tiempo la tecla **Ctrl**. Para finalizar suelte el botón.

Para *escribir texto*, primero haga clic sobre este botón. A continuación sitúe el cursor del ratón en la posición del panel donde se desea que comience el texto, pulse el botón izquierdo del mismo y escriba el texto.

Inicialmente, la utilidad por defecto es la de *dibujar a mano alzada*. Posteriormente, la elección de cualquier otra herramienta cancela la anteriormente se-

leccionada. Si hace clic sobre el botón que está pulsado, no se ejecutará ninguna acción.

Nuestro primer paso para el desarrollo de la aplicación es crear los iconos necesarios para la misma, que se muestran a continuación.

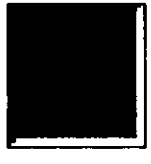
- Dibujar a mano alzada.

MANO_OF.ICO

MANO_ON.ICO

- Dibujar líneas.

LIN_OF.ICO



LIN_ON.ICO

- Dibujar cajas.

CAJ_OF.ICO

CAJ_ON.ICO

- Dibujar círculos o elipses.

CIR_OF.ICO

CIR_ON.ICO

- Escribir texto.

TEXT_OF.ICO

TEXT_ON.ICO

Para crear los iconos, cargue y ejecute la aplicación *iconwrks*, que es un editor de iconos que se suministra como ejemplo con Visual Basic. Abra el fichero PENCIL05.ICO que se encuentra en el directorio "...\vb\icons\writing", haga las modificaciones necesarias para que tome el aspecto de la figura anterior y guárdelo en su directorio de trabajo como MANO_OF.ICO. De la misma forma y a partir del icono creado, diseñe el icono MANO_ON.ICO. El resto de los iconos serán construidos, utilizando las herramientas del editor y tomando como plantilla los ya creados. El resultado será similar al presentado en las figuras anteriores.

Inicie una nueva aplicación, asigne a la forma por defecto el título *Panel de Dibujo* y demás propiedades que se especifican en la tabla siguiente, ajuste su tamaño y guarde la aplicación con el nombre *panel.mak* y la forma con el nombre *panel.frm*.

Objeto	Propiedad	Valor
Forma	FormName Caption BorderStyle Icon	Form1 Panel de Dibujo 2 DIBUJO.ICO

La propiedad Icon de la forma permite que se visualice el icono asociado a la misma cuando la forma se minimiza. Para nuestra aplicación, hemos elegido el icono PENCIL03.ICO que se encuentra en el directorio "...\vb\icons\writing". Cargue el editor de iconos y realice las modificaciones oportunas, si lo considera necesario, y a continuación guárdelo en su directorio de trabajo con el nombre DIBUJO.ICO.

Continuando con la aplicación, diseñe un menú denominado *Fichero* con las órdenes que se indican en la tabla siguiente.

Caption	CtlName	Acelerator
&Nuevo	FicheroNuevo	Ctrl+N
&Salir	FicheroSalir	Ctrl+S

El siguiente procedimiento da lugar a que, cuando el usuario ejecute la orden *Nuevo*, se limpie el área de dibujo.

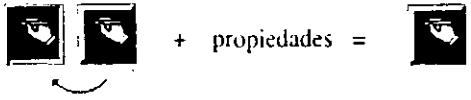
```
Sub FicheroNuevo_Click ()
  Cls
End Sub
```

El siguiente procedimiento da lugar a que, cuando el usuario ejecute la orden *Salir*, finalice la aplicación.

```
Sub FicheroSalir_Click ()
  End
End Sub
```

A continuación vamos a construir el panel de control. Para ello añada los controles que se indican en la tabla siguiente.

La figura siguiente le indica cómo debe añadir cada pareja de iconos. Primero coloca una pareja de iconos uno al lado de otro, a continuación asigna las propiedades especificadas y por último coloca el icono ...OF.ICO sobre el icono ...ON.ICO. Observe que los iconos ...ON.ICO son no visibles inicialmente.



Siguiendo el mismo proceso, coloque el resto de las parejas una debajo de otra hasta completar el panel de utilidades.

Objeto	Propiedad	Valor
Caja de imagen	CtlName	Botón
	AutoSize	True
	BorderStyle	1
	Index	0
	Picture	MANO_ON.ICO
	Visible	False

Caja de imagen	CtlName	Botón
	AutoSize	True
	BorderStyle	1
	Index	1
	Picture	MANO_OF.ICO
	Visible	True
Caja de imagen	CtlName	Botón
	AutoSize	True
	BorderStyle	1
	Index	2
	Picture	LIN_ON.ICO
	Visible	False
Caja de imagen	CtlName	Botón
	AutoSize	True
	BorderStyle	1
	Index	3
	Picture	LIN_OF.ICO
	Visible	True
Caja de imagen	CtlName	Botón
	AutoSize	True
	BorderStyle	1
	Index	4
	Picture	CAJ_ON.ICO
	Visible	False
Caja de imagen	CtlName	Botón
	AutoSize	True
	BorderStyle	1
	Index	5
	Picture	CAJ_OF.ICO
	Visible	True
Caja de imagen	CtlName	Botón
	AutoSize	True
	BorderStyle	1
	Index	6
	Picture	CIR_ON.ICO
	Visible	False
Caja de imagen	CtlName	Botón
	AutoSize	True
	BorderStyle	1
	Index	7
	Picture	CIR_OF.ICO
	Visible	True

Caja de imagen	CtlName	Botón
	AutoSize	True
	BorderStyle	1
	Index	8
	Picture	TEXT_ON.ICO
	Visible	False
Caja de imagen	CtlName	Botón
	AutoSize	True
	BorderStyle	1
	Index	9
	Picture	TEXT_OF.ICO
	Visible	True

Las cajas de imagen que forman el panel de utilidades constituyen un array de controles que hemos denominado *Botón*.

Para completar nuestro panel de dibujo añada las etiquetas indicadas en la tabla siguiente.

Objeto	**Propiedad**	**Valor**
Etiqueta	CtlName	Label1
	Caption	X:
	ForeColor	&H00000000&
	BackColor	&H00FFFFFF&
Etiqueta	CtlName	X
	Caption	(nada)
	ForeColor	&H00000000&
	BackColor	&H00FFFFFF&
Etiqueta	CtlName	Label2
	Caption	Y:
	ForeColor	&H00000000&
	BackColor	&H00FFFFFF&
Etiqueta	CtlName	Y
	Caption	(nada)
	ForeColor	&H00000000&
	BackColor	&H00FFFFFF&

El siguiente paso es añadir el código necesario para que nuestra aplicación realice las funciones especificadas anteriormente. Primeramente vamos a hacer funcionar el panel de utilidades. Recuerde que sólo puede haber un botón pulsado

cada vez. Cuando pulse cualquier otro botón (hacer clic), el botón que está pulsado volverá automáticamente a su posición normal (sin pulsar). Si hace clic sobre el botón que está pulsado, no se ejecutará ninguna acción. Por lo tanto, el procedimiento que realice este proceso responderá al suceso **Click**.

Según la tabla empleada para construir el panel de utilidades, un botón sin pulsar tiene índice impar (**Index** = 1, 3, 5, 7 o 9) y un botón pulsado tiene índice par (0, 2, 4, 6 u 8). De acuerdo con esto, el procedimiento *Botón_Click*, primero realiza la acción de pulsar el botón y después restaura el botón que estaba pulsado a su posición normal: el índice de éste botón se guarda de una vez para la siguiente en la variable *IndAnt* definida a nivel de la forma. El efecto pulsado/no pulsado de un botón, se consigue a través de dos imágenes ...ON/...OF. Pulsar un botón equivale a hacer visible la imagen ...ON y hacer no visible la imagen ...OF. Lo contrario vuelve el botón a su posición normal (sin pulsar).

```
Sub Botón_Click (Index As Integer)
  'Pulsar el botón Index
  Select Case Index
    Case 1
      Botón(0).Visible = -1: Botón(1).Visible = 0
    Case 3
      Botón(2).Visible = -1: Botón(3).Visible = 0
    Case 5
      Botón(4).Visible = -1: Botón(5).Visible = 0
    Case 7
      Botón(6).Visible = -1: Botón(7).Visible = 0
    Case 9
      Botón(8).Visible = -1: Botón(9).Visible = 0
    Case Else
      Exit Sub
  End Select
  'Al pulsar un botón sale el anterior
  'IndAnt puede se 1, 3, 5, 7 o 9
  Botón(IndAnt - 1).Visible = 0
  Botón(IndAnt).Visible = -1
  IndAnt = Index
End Sub
```

Ahora ya tenemos en marcha el panel de utilidades y sabemos que un botón pulsado tiene índice par y que su propiedad visible vale −1.

Para continuar defina a nivel de la forma (**(general)**) las variables que se indican a continuación y que explicaremos según aparezcan.

```
Dim IndAnt As Integer, CX As Single, CY As S    e
Dim Radio As Single, Aspecto As Single
```

Cuando se arranque la aplicación, deseamos como color de fondo el blanco y como color del primer plano (dibujo) el negro. También aparecerá como opción por defecto, dibujar a mano alzada. El procedimiento *Form_Load* escrito a continuación realiza estas operaciones. Para conseguir que inicialmente aparezca pulsado el botón de dibujo a mano alzada, invocamos al procedimiento *Botón_Click* con un valor de 1 para **Index** y como este procedimiento lo último que hace es retornar el botón que estaba pulsado a su posición normal, hacemos el supuesto que el que lo estaba era el botón para escribir texto (*IndAnt* = 9). A pesar de que inicialmente esto no es así, el resultado deseado no se ve alterado.

```
Sub Form_Load ()
  BackColor = QBColor(15) 'fondo blanco
  ForeColor = QBColor(0)  'dibujo en negro
  IndAnt = 9
  Botón_Click 1
End Sub
```

Como ya explicamos, al mover el ratón sobre un control se da el suceso **MouseMove** para ese control. Pues bien, cuando arrastremos el cursor del ratón por encima de los botones que forman el panel de utilidades, queremos que éste tenga forma de flecha. Como el procedimiento invocado en este caso es *Botón_MouseMove*, lo único que tenemos que hacer es incluir en el cuerpo del mismo la sentencia *Screen.MousePointer = 0*.

```
Sub Botón_MouseMove (Index As Integer, Button As
  Integer, Shift As Integer, X As Single, Y As Single)
  Screen.MousePointer = 0
End Sub
```

Una vez que el usuario haya elegido la utilidad deseada del panel, lo primero que hará será situar el cursor del ratón en el lugar donde desee comenzar el dibujo y pulsar el botón izquierdo del mismo (*Button = 1*). Esto hace que se dé el suceso **MouseDown** y como respuesta, almacenaremos las coordenadas actuales del cursor del ratón en las variables *CX* y *CY* y en las propiedades **CurrentX** y **CurrentY**. También es necesario, como veremos mas adelante al explicar la rutina de dibujar elipses y círculos, poner la variable *Radio* a valor 0.

```
Sub Form_MouseDown (Button As Integer, Shift As
                    Integer, X As Single, Y As Single)
  If Button = 1 Then
    CX = X: CurrentX = X
    CY = Y: CurrentY = Y
    Radio = 0
  End If
End Sub
```


Para realizar un dibujo, el usuario arrastrará el ratón con el botón izquierdo pulsado, lo que hace que se dé el suceso **MouseMove**. La respuesta a este suceso depende del botón del panel que haya pulsado. Por otra parte, cuando arrastremos el cursor del ratón por encima del área de dibujo, esto es sobre la forma, queremos que tenga forma de cruz, lo cual se consigue ejecutando la sentencia *Screen.MousePointer = 2* en el procedimiento *Form_MouseMove*. También, como respuesta a este suceso, actualizaremos las etiquetas *X* e *Y* que muestran la posición actual del cursor del ratón.

El procedimiento que se presenta a continuación, realiza las operaciones anteriormente expuestas y las rutinas correspondientes a cada uno de los botones del panel. Es importante que recuerde la función de las propiedades **CurrentX** y **CurrentY** para cada tipo de gráfico, vistas en el Capítulo 9.

El proceso de dibujar comienza si el botón izquierdo del ratón se mantiene pulsado (*Button = 1*). Si además el botón del panel pulsado es el de dibujar a mano alzada, arrastrando el ratón puede realizar esta función (*Line -(X, Y)*). Si es el botón de dibujar líneas, cajas o elipses (*If Not Botón(0).Visible*) es necesario que cada una de sus rutinas asociadas permita variar la figura que estamos dibujando hasta obtener la que deseamos, momento en que soltaremos el botón izquierdo del ratón. Este efecto se consigue dibujando con la pluma **Xor** (*DrawMode = 7*).

Pantalla	Pluma	Pantalla Xor Pluma
0	0	0
0	1	1
1	0	1
1	1	0

La forma sobre la que se dibuja es de color blanco (todo unos). Por lo tanto, si dibujamos con color blanco (**ForeColor = BackColor**) obtendremos un dibujo en negro (todo ceros). Para borrar este dibujo, lo dibujamos de nuevo justamente encima; el resultado es el dibujo en blanco (color de fondo), esto es, no se visualiza. Esta es la técnica utilizada para conseguir el efecto al que nos referíamos en el párrafo anterior.

En resumen, el efecto de variar una figura dinámicamente se consigue borrando la figura existente y dibujando una nueva. La posición de la línea o caja existente es conocida por las coordenadas (*CX, CY*) y (**CurrentX, CurrentY**). Una elipse o círculo existente queda definida/o por las coordenadas (*CX, CY*) y por las variables *Radio* y *Aspecto*. Como estas variables son estáticas, conservan

su valor de una vez para la siguiente. En las figuras nuevas además del punto inicial (*CX, CY*) interviene el punto actual (*X, Y*). Por ejemplo,

```
Line (CX, CY)-(CurrentX, CurrentY) 'borrar línea
Line (CX, CY)-(X, Y) 'dibujar nueva línea
```

Inicialmente, la primera sentencia de la pareja que hay en cada rutina de dibujo no ejecuta ninguna acción, por coincidir las coordenadas iniciales y finales, o en el caso de las elipses y círculos por ser el radio 0. Cuando se dibuja una línea o una caja, automáticamente las coordenadas finales (*X, Y*) pasan a almacenarse en las propiedades (**CurrentX**, **CurrentY**).

```
Sub Form_MouseMove (Button As Integer, Shift As
                    Integer, X As Single, Y As Single)
  Dim DMInicial As Integer, ColorIni As Long

  'Coordenadas X, Y actuales
  Form1.X.Caption = Str$(X - Botón(0).Width)
  Form1.Y.Caption = Str$(Y)

  'Puntero en cruz
  Screen.MousePointer = 2

  If Button = 1 Then 'Si pulsado botón izdo. del ratón
    'Dibujar a mano alzada
    If Botón(0).Visible Then
      Line -(X, Y)
    End If

    If Not Botón(0).Visible Then
      ColorIni = ForeColor
      ForeColor = BackColor
      DMInicial = DrawMode
      DrawMode = 7  'Xor

      'Dibujar líneas
      If Botón(2).Visible Then
        Line (CX, CY)-(CurrentX, CurrentY)
        Line (CX, CY)-(X, Y)
      End If

      'Dibujar cajas
      If Botón(4).Visible Then
        Line (CX, CY)-(CurrentX, CurrentY), , B
        If Shift = 2 Then
          Line (CX, CY)-(CX - (CY - Y), Y), , B
        End If
      End If
      Line (CX, CY)-(X, Y), , B
```



```
      End If
    End If

    'Dibujar círculos
    If Botón(6).Visible Then
      'Radio y Aspecto son variables estáticas
      Circle (CX, CY), Radio, , , , Aspecto
      Radio = Sqr((CX - X) ^ 2 + (CY - Y) ^ 2)
      If (CX - X) <> 0 Then
        Aspecto = (CY - Y) / (CX - X)
      End If
      If Shift = 2 Then Aspecto = 1
      Circle (CX, CY), Radio, , , , Aspecto
    End If

    DrawMode = DMInicial
    ForeColor = ColorIni
  End If
 End If
End Sub
```

Observe que en la rutina de dibujar cajas, si mantenemos pulsada la tecla **Ctrl** (*Shift* = 2), se dibujan cuadrados perfectos y en caso de las elipses, círculos perfectos (*Aspecto* = *1*).

Una vez finalizado el dibujo de una línea, caja o elipse, se restauran los valores iniciales de las propiedades **DrawMode** y **ForeColor**.

Cuando el usuario hace clic en el botón del panel que permite escribir texto, dicho botón queda enfocado. Por lo tanto, cuando pulse una tecla para escribir un carácter se producirá el suceso **KeyPress** para ese control (*Botón*). Quiere esto decir que en el procedimiento *Botón_KeyPress* es donde tenemos que colocar el código que permite escribir cada carácter pulsado. Para realizar esto utilizaremos el método **Print**. La escritura comienza en la posición indicada por las propiedades **CurrentX** y **CurrentY**.

Cuando **Print** escribe una expresión, ésta es finalizada con un retorno de carro. Entonces la propiedad **CurrentX** es puesta a cero y la propiedad **CurrentY** es incrementada en el alto de la expresión (**TextHight**). Si a continuación de la expresión colocamos un punto y coma, una vez escrita la misma no se produce un retorno de carro. En este caso la propiedad **CurrentX** es incrementada en el ancho de la expresión (**TextWidth**) y la propiedad **CurrentY** no cambia.

```
Sub Botón_KeyPress (Index As Integer, KeyAscii As Integer)
  Form1.Print Chr$(KeyAscii);
End Sub
```

SELECCIONAR Y ARRASTRAR

Cuando estamos diseñando una aplicación Visual Basic, a menudo seleccionamos un control y lo arrastramos a la posición deseada. Esta característica puede ser puesta a disposición del usuario de una aplicación, en tiempo de ejecución. La forma de ejecutar una acción como ésta es hacer clic en el objeto y con el botón izquierdo del ratón pulsado, arrastrarlo hacia la posición deseada.

Cuando un control se arrastra hacia una nueva posición durante la ejecución de una aplicación, nosotros mismos deberemos programar cual será esa nueva posición. No obstante, esta característica se utiliza a menudo para permitir que alguna otra acción se ejecute: en este caso, al soltar el botón, el objeto vuelve automáticamente a su posición inicial.

Para permitir al usuario seleccionar un control y arrastrarlo, asigne a la propiedad **DragMode** del control el valor 1. Esta operación puede hacerla durante el diseño o durante la ejecución. Por ejemplo,

```
Picture1.DragMode = 1
```

Cuando se arrastra un control, Visual Basic muestra un cuadro gris en lugar del propio control. Si quiere, puede sustituir este cuadro por otra imagen, durante el diseño, asignando el fichero correspondiente a la propiedad **DragIcon**, o durante la ejecución como se indica en los ejemplos siguientes.

```
Picture1.DragIcon = Picture2.DragIcon
Picture1.DragIcon = Picture2.Picture
Picture1.DragIcon = LoadPicture("c:\vb\icons\arrows\arr04rt")
```

Cuando el usuario suelta el botón para dejar de arrastrar un objeto, se genera el suceso **DragDrop** para el objeto sobre el cual se arrastra. El esquema del procedimiento correspondiente a este suceso cuando el control se arrastra sobre una forma es,

```
Sub Form_DragDrop (Source As Control,
                   X As Single, Y As Single)

End Sub
```

El argumento *Source* referencia el control que se arrastra, el cual puede ser de cualquier tipo excepto un menú o un temporizador. La palabra *Form* que aparece en el nombre del procedimiento *Form_DragDrop* especifica el objeto sobre el cual se arrastra; en general puede ser una forma o un control.

Los argumentos *X*, *Y*, son las coordenadas correspondientes al puntero del ratón, referidas al objeto (forma o control) sobre el que se arrastra.

Mientras la propiedad de arrastrar automáticamente un control esté activa, éste no reconoce otros sucesos del ratón. Dicho de otra forma, como la operación de arrastrar empieza con un clic sobre el control, los sucesos del ratón, por ejemplo **MouseDown**, no son reconocidos por el control.

Como ejemplo, cree un procedimiento para que la aplicación que diseñamos anteriormente en este mismo capítulo y que denominamos *raton.mak*, permita arrastrar el icono CIR_OF.ICO a una nueva posición. Para realizar este proceso, cargue la aplicación, seleccione el control denominado *Picture1* que contiene el icono y ponga su propiedad **DragMode** al valor 1 (arrastrar automáticamente).

Cuando el usuario seleccione y arrastre el objeto sobre la forma (*Form*), en el momento que suelte el botón para dejarlo en la posición alcanzada, se genera el suceso **DragDrop**. Si no hay una respuesta a este suceso, el objeto retorna automáticamente a su posición inicial. Para colocar el objeto en la nueva posición, responda a este suceso con el método **Move** y las coordenadas correspondientes al puntero del ratón.

```
Sub Form_DragDrop (Source As Control,
                   X As Single, Y As Single)
  CXP = Picture1.Width / 2
  CYP = Picture1.Height / 2
  Picture1.Move X - CXP, Y - CYP
End Sub
```

Con el fin de que el centro del objeto sea el que se sitúe en la posición indicada por las coordenadas (*X*, *Y*) del puntero del ratón y no la esquina superior izquierda del mismo, se resta a *X* una cantidad igual a la mitad del ancho del objeto y a *Y* una cantidad igual a la mitad del alto.

Cuando una operación de seleccionar y arrastrar está en proceso y el puntero del ratón entra en, sale de, o está encima de, un control cualquiera de la forma, se da el suceso **DragOver** asociado con dicho control. Por ejemplo, si el control sobre el que se arrastra es una etiqueta *Label1*, el esquema del procedimiento para este suceso es el siguiente:

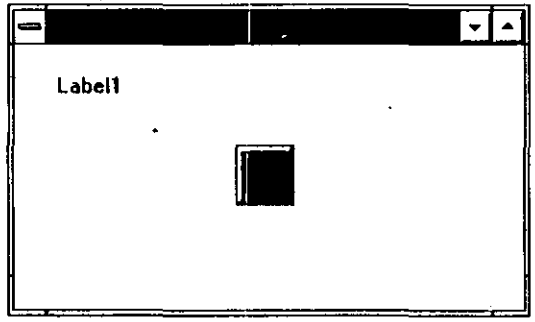
```
Sub Label1_DragOver (Source As Control, X As Single,
                     Y As Single, State As Integer)

End Sub
```

Cuando se da el suceso **DragOver** aparece un nuevo parámetro *State*. Si este parámetro toma el valor 0 es porque el puntero del ratón entra en el objeto, en nuestro caso en *Label1*; si toma el valor 1 es porque el puntero del ratón deja o sale del objeto; y si toma el valor 2 es porque el control *Source* está encima del objeto.

Normalmente, los estados 0 (entrar) y 1 (salir) se utilizan para distinguir cuando estamos dentro del área de un objeto, mientras que el estado 2 es útil para saber si el puntero del ratón está encima de una determinada zona de un objeto. En este último caso, será necesario evaluar los argumentos *X* e *Y*.

Como ejemplo, añada una etiqueta *Label1* a la forma y ponga la propiedad **Caption** de la misma a un valor nulo.

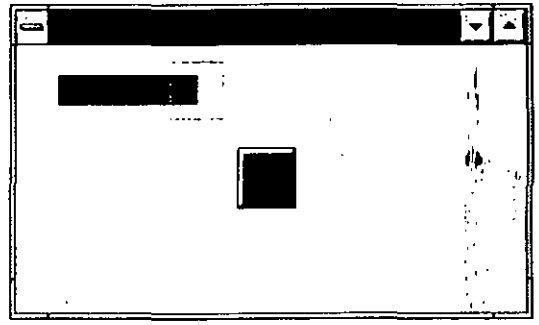
A continuación, cree un procedimiento para dicha etiqueta, de forma que si al arrastrar el control *Picture1* (imagen CIR_OF.ICO) entra en el área correspondiente a *Label1*, se inviertan los colores del fondo y del primer plano de la misma y lo mismo cuando salga de la etiqueta.



Abra la ventana de código correspondiente a *Label1* y escriba el siguiente procedimiento.

```
Sub Label1_DragOver (Source As Control, X As Single,
                     Y As Single, State As Integer)
  Const ENTRA = 0
  Const SALE = 1
  Dim ColorTemp As Long
  If State = ENTRA Or State = SALE Then
    ColorTemp = Label1.BackColor
    Label1.BackColor = Label1.ForeColor
    Label1.ForeColor = Color
      If
End   b
```


Guarde y ejecute la aplicación.



Observe que al arrastrar el control *Picture1*, si entra en la etiqueta, ésta invierte su color y cuando sale de ella ocurre lo mismo.

A continuación, modifique el procedimiento *Form_DragDrop* para que permita también arrastrar la etiqueta. La solución se presenta a continuación.

```
Sub Form_DragDrop (Source As Control, X As Single, Y
As Single)
  If TypeOf Source Is PictureBox Then
    CX = Picture1.Width / 2
    CY = Picture1.Height / 2
    Picture1.Move X - CX, Y - CY
  ElseIf TypeOf Source Is Label Then
    CX = Label1.Width / 2
    CY = Label1.Height / 2
    Label1.Move X - CX, Y - CY
  End If
End Sub
```

Observe que para saber de qué objeto se trata utilizamos la expresión,

If TypeOf *objeto* **Is** *tipo_objeto*

El argumento *objeto* es una variable de tipo **Control** y *tipo_objeto* indica el tipo del objeto en cuestión, el cual puede ser uno de los siguientes: CheckBox, ComboBox, CommandButton, DirListBox, DriveListBox, FileListBox, Frame, HScrollBar, Label, ListBox, Menu, OptionButton, PictureBox, TextBox, Timer, o VScrollBar.

Hasta ahora hemos visto que cuando la propiedad **DragMode** determinado control tiene un valor 1, dicho control puede ser arrastrado a icamente y no reconoce otros sucesos del ratón. Además de la forma aut ica, Visual

Basic tiene la forma manual. propiedad **DragMode** = 0, que permite controlar cuando un objeto puede ser arrastrado y cuando no. Por ejemplo. usted puede querer arrastrar un objeto en respuesta a un suceso **MouseDown**. a una tecla. o a una orden de un menú.

Cuando la propiedad **DragMode** vale 0 (manual). la forma de permitir o no arrastrar un objeto es utilizando el método **Drag** cuya sintaxis es la siguiente:

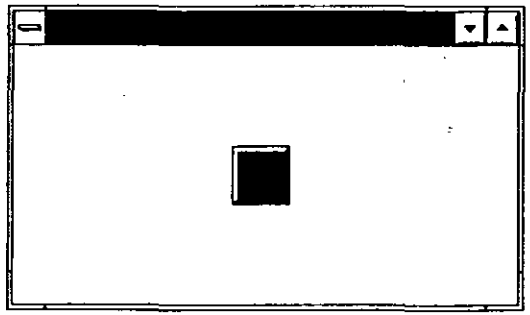
[*control.*]**Drag** *acción*

El valor del argumento *acción* puede ser 0, 1 o 2. Un valor 0. cancela la operación de arrastrar; un valor 1, inicia la operación de arrastrar: un valor 2. finaliza la operación de arrastrar y además hace que se dé el suceso **DragDrop**.

Como ejemplo cree una nueva aplicación llamada *manual.mak*. Cree una nueva forma llamada *manual.frm* y añada a la misma los controles de la tabla siguiente:

Objeto	Propiedad	Valor
Caja de imagen	CtlName AutoSize DragMode Picture	Picture1 True 0 - Manual CIR_OF.ICO

La forma será similar a la de la figura siguiente.



Nuestro objetivo es que cuando el usuario seleccione y arrastre la imagen *Picture1*. el control se mueva precisamente dentro de la posición final del rectángulo gris. Para conseguir esto, hay que registrar las coordenadas iniciales correspondientes a la posición del ratón dentro del control. Entonces cuando se mueva el objeto utilizaremos esta posición como un desplazamiento.

Los pasos a seguir para realizar lo especificado son los siguientes:

- Declarar dos variables a nivel de la forma *DespX* y *DespY*.

```
Dim DespX As Single, DespY As Single
```

- Permitir arrastrar el control cuando ocurra el suceso **MouseDown**. También, almacenar los valores de *X* y de *Y* en *DespX* y *DespY* respectivamente.

```
Sub Picture1_MouseDown (Button As Integer, Shift As
                        Integer, X As Single, Y As Single)
  Picture1.Drag 1 'inicio arrastre
  DespX = X
  DespY = Y
End Sub
```

- Cuando finaliza el arrastre y se suelta el botón del ratón, se produce el suceso **DragDrop**. El procedimiento asociado con la forma para este suceso, situará el control en la posición (*X*– *DespX*, *Y*– *DespY*).

```
Sub Form_DragDrop (Source As Control, X As Single,
                   Y As Single)
  Picture1.Move X - DespX, Y - DespY
End Sub
```

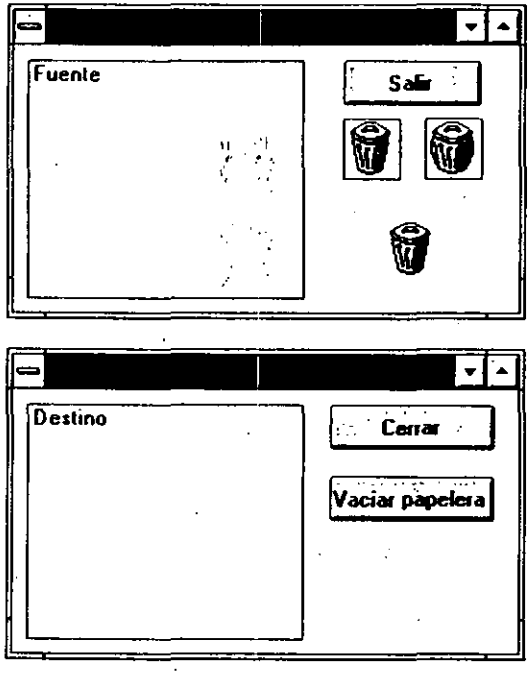
EJEMPLO DE SELECCIONAR Y ARRASTRAR

El siguiente ejemplo emula el funcionamiento de la papelera en un sistema Macintosh. En un sistema como éste. para borrar un fichero que aparece en una lista *Fuente*, simplemente se selecciona y se arrastra hasta la papelera. donde se deja caer. La papelera para indicar que tiene algo. aumentará de volumen. Para borrar cualquier otro fichero de la lista, se procede de igual forma..

Estos ficheros que se han eliminado de la lista *Fuente* y que se encuentran en la papelera, aún no están físicamente borrados de la unidad de almacenamiento correspondiente. Para ver lo que hay en la papelera haga un doble clic sobre la misma y aparecerá una segunda ventana. Observe las dos figuras siguientes.

La lista *Destino* indica los ficheros que hay en la papelera. Cualquiera de estos ficheros puede ser devuelto a la lista *Fuente* seleccionándolo y arrastrándolo hasta dejarlo caer sobre la misma; este proceso equivale a recuperar el fichero. Para bo-

rrar definitivamente los ficheros que se encuentran en la papelera se pulsa el botón *Vaciar papelera*.



Nuestro primer paso es crear los iconos necesarios para nuestra aplicación: papelera vacía (PAPELERA.ICO), papelera oscura vacía (PAPEOSVA.ICO), papelera llena o con algún fichero (PAPELLE.ICO), papelera oscura llena o con algún fichero (PAPEOSLL.ICO) y un icono indicando que hemos seleccionado un fichero y lo trasladamos de lugar (FICHERO.ICO).

Para crear los iconos, cargue y ejecute la aplicación *iconwrks*, que es un editor de iconos que se suministra como ejemplo con Visual Basic. Abra el fichero TRASH01.ICO que se encuentra en el directorio "...\vb\icons\computer", haga las modificaciones que considere necesarias y guárdelo en su directorio de trabajo como PAPELERA.ICO. Proceda de forma similar para las otras tres papeleras. El resultado será similar al presentado en la figura siguiente.

PAPELERA.ICO PAPEOSVA.ICO

PAPELLE.ICO PAPEOSLL.ICO

Para crear el quinto icono abra el fichero FILES10.ICO que se encuentra en el directorio "...\vb\icons\office", haga las modificaciones para que quede como se muestra en la figura siguiente y guárdelo en su directorio de trabajo como FICHERO.ICO.

FICHERO.ICO

Inicie una nueva aplicación y cree una forma titulada *Tirar en la papelera* con los controles que se indican en la tabla siguiente (ver en las páginas anteriores la correspondiente figura). Guarde la aplicación con el nombre *papelera.mak* y la forma con el nombre *papele01.frm*.

Objeto	Propiedad	Valor
Forma	FormName Caption BorderStyle ControlBox Icon	Form1 Tirar en la papelera 1 False PAPELERA.ICO
Lista	CtlName DragIcon Sorted Tag	Fuente FICHERO.ICO True Fuente

Botón	CtlName Caption	Salir Salir
Caja de imagen	CtlName AutoSize BorderStyle DragIcon Picture Visible	Vacía True 1 PAPEOSVA.ICO PAPELERA.ICO False
Caja de imagen	CtlName AutoSize BorderStyle DragIcon Picture Visible	Llena True 1 PAPEOSLL.ICO PAPELLE.ICO False
Caja de imagen	CtlName AutoSize BorderStyle Picture Visible	Papelera True 0 PAPELERA.ICO False

La propiedad **ControlBox** de la forma permite habilitar (**True**) o deshabilitar (**False**) el menú de control de dicha forma. La propiedad **Icon** de la forma permite que se visualice el icono asociado a la misma cuando la forma se minimiza.

La propiedad **Tag** de un objeto permite almacenar un único dato, que será utilizado para identificar el objeto. Otras veces, esta propiedad se utiliza con fines de almacenamiento.

Las cajas de imagen nombradas *Vacía* y *Llena*, que durante la ejecución permanecen ocultas, sirven para proveer a la caja de imagen *Papelera* de la imagen adecuada en cada momento. Observe que cada una de ellas almacena dos imágenes, la real en la propiedad **Picture** y la oscurecida en la propiedad **DragIcon**. En este caso, la propiedad **DragIcon** se ha utilizado con fines de almacenamiento y no con intención de arrastrar. También, observe que la caja de imagen *Papelera*, inicialmente muestra la papelera vacía.

Cree una nueva forma titulada *Recuperar o vaciar* con los controles que se indican en la tabla siguiente (ver en las páginas anteriores la correspondiente figura). A continuación, guarde la aplicación con el nombre que ya tiene y la forma con el nombre *papele02.frm*.

Objeto	**Propiedad**	**Valor**
Forma	FormName Caption BorderStyle ControlBox Icon	Form2 Recuperar o vaciar 1 False PAPELERA.ICO
Lista	CtlName Sorted Tag	Destino True Destino
Botón	CtlName Index Caption	Orden 0 Cerrar
Botón	CtlName Index Caption	Orden 1 Vaciar papelera

En primer lugar escriba el código asociado con el botón *Salir* para el suceso **Click**. El objetivo de este procedimiento es finalizar la aplicación.

```
Sub Salir_Click ()
  End
End Sub
```

Para hacer más sencilla la aplicación, en lugar de que un sistema de ficheros muestre la lista de ficheros de un directorio elegido, vamos a generar nosotros una lista que emule a la citada.

El siguiente procedimiento, que se ejecutará cuando se inicie la aplicación, genera la lista de elementos para demostración y además asigna a la lista *Destino*, el icono que se visualiza cuando se selecciona y arrastra la lista *Fuente*, con el mismo fin. Por último se asigna a la propiedad **ListIndex** de la lista *Fuente* el valor 0 lo que hará que quede seleccionado automáticamente el primer elemento de la misma.

```
Sub Form_Load ()
  Dim I As Integer
  Form2.Destino.DragIcon = Fuente.DragIcon
  For I = 1 To 15
    Fuente.AddItem "Fuente_" + Format$(I, "00")
  Next
  Fuente.ListIndex = 0 'Seleccionar primer elemento
End Sub
```

Una lista responde a un clic, realizado con el botón izquierdo (**Button** = 1) sobre un elemento de la misma, cambiando de color el elemento seleccionado y poniendo su propiedad **ListIndex** al valor correspondiente a la posición del elemento en la lista (el primer elemento ocupa la posición 0). El número de elementos de la lista lo da la propiedad **ListCount**.

Nuestro siguiente objetivo es arrastrar la lista, manteniendo pulsado el botón derecho del ratón (**Button** = 2), y dejar caer en la papelera el elemento seleccionado. Cuando se pulsa un botón del ratón se da el suceso **MouseDown**. Si en la lista hay elementos y el botón pulsado es el derecho, iniciamos el proceso de arrastre de la lista *Fuente*.

Observe que no se ha utilizado el arrastre automático ya que una vez iniciada la operación de arrastrar, el control *Fuente* no reconocería otros sucesos del ratón. Esto es, al hacer clic sobre el objeto para iniciar el arrastre, los sucesos del ratón, como por ejemplo **MouseDown**, no serían reconocidos. Esto, por ejemplo, implicaría no poder seleccionar un elemento.

```
Sub Fuente_MouseDown (Button As Integer, Shift As
                      Integer, X As Single, Y As Single)
  If Fuente.ListCount > 0 Then 'si lista no vacía
    If Button = 2 Then 'si botón derecho pulsado
      Fuente.Drag 1 'inicio arrastre
    End If
  End If
End Sub
```

El elemento seleccionado lo dejaremos caer en la papelera cuando estemos sobre ella y soltemos, en nuestro caso, el botón derecho del ratón. En el momento de soltar el botón derecho del ratón el arrastre finaliza y se da el suceso **DragDrop** para el control sobre el que se estaba arrastrando y cuando en una operación de arrastre se entra en un control, se da el suceso **DragOver**. Este último suceso lo utilizaremos para indicar al usuario cuando entra en, y cuando sale de, la papelera. Cuando entre, cambiaremos la imagen de la papelera a oscura y cuando salga volveremos a poner la imagen real.


```
Sub Papelera_DragOver (Source As Control, X As Single,
Y As Single, State As Integer)
  Select Case State
    Case 0 'entra en papelera
      If Form2.Destino.ListCount = 0 Then
        Papelera.Picture = Vacía.DragIcon
      Else
        Papelera.Picture = Llena.DragIcon
      End If
    Case 1 'sale de papelera
      If Form2.Destino.ListCount = 0 Then
        Papelera.Picture = Vacía.Picture
      Else
        Papelera.Picture = Llena.Picture
      End If
  End Select
End Sub
```

Según hemos dicho, en el momento de soltar el botón derecho del ratón se da el suceso **DragDrop** que lo utilizaremos para mover el elemento seleccionado de la lista *Fuente*, a la lista *Destino*. El elemento seleccionado lo dejaremos caer sobre la papelera, sólo si el control arrastrado (parámetro *Source*) es la lista *Fuente*. Cuando se trate de otro control, por ejemplo lista *Destino*, se realizará un aviso acústico. La operación mover implica añadir el elemento seleccionado de la lista *Fuente* a la lista *Destino* y borrarlo de la lista *Fuente*. También, si no hay un elemento seleccionado en la lista *Destino* (**ListIndex** = −1), se selecciona el elemento 0 de la misma. Por último, se cambia la apariencia de la papelera para que muestre la imagen real de llena, que estaba vacía o llena pero oscurecida.

```
Sub Papelera_DragDrop (Source As Control, X As Single,
                       Y As Single)
  If Source.tag <> "Destino" Then
    Form2.Destino.AddItem Source.List(Source.ListIndex)
    If Form2.Destino.ListIndex = -1 Then
      Form2.Destino.ListIndex = 0
    End If
    BorrarElemento Source
  Else
    Beep
  End If
  If Papelera.Picture <> Llena.Picture Then
    Papelera.Picture = Llena.Picture
  End If
End Sub
```

Para ver el contenido de la lista *Destino* hay que hacer un doble clic en la papelera.

```
Sub Papelera_DblClick ()
  Form2.Show
End Sub
```

Cuando la lista destino está presente en la pantalla, se permitirá al usuario mover un elemento directamente desde la lista *Fuente* a la lista *Destino*. Esto es, en lugar de dejar caer el elemento en la papelera, lo dejará caer directamente en la lista *Destino*, obteniendo los mismos resultados. Por lo tanto, el procedimiento *Destino_DragDrop* es muy similar al procedimiento *Papelera_DragDrop* anteriormente explicado.

```
Sub Destino_DragDrop (Source As Control, X As Single,
                      Y As Single)
  If Source.Tag <> "Destino" Then
    If Form1.Papelera.Picture <> Form1.Llena.Picture Then
      Form1.Papelera.Picture = Form1.Llena.Picture
    End If
    Destino.AddItem Source.List(Source.ListIndex)
    If Destino.ListIndex = -1 Then
      Destino.ListIndex = 0
    End If
    BorrarElemento Source
  End If
End Sub
```

Otra operación a tener en cuenta es que el usuario pueda recuperar un elemento borrado de la lista *Fuente*, que se encuentre en la lista *Destino*. Este proceso consiste en arrastrar la lista *Destino*, manteniendo pulsado el botón derecho del ratón (**Button** = 2), y dejar caer en la lista *Fuente* el elemento seleccionado. Si en la lista *Destino* hay elementos y el botón pulsado es el derecho, iniciamos el proceso de arrastre.

```
Sub Destino_MouseDown (Button As Integer, Shift As
                       Integer, X As Single, Y As Single)
  If Destino.ListCount > 0 Then
    If Button = 2 Then
      Destino.Drag 1
    End If
  End If
End Sub
```

Como ya sabe, al soltar el botón derecho del ratón para finalizar el arrastre y dejar caer el objeto, se da el suceso **DragDrop** que lo utilizaremos para mover el elemento seleccionado de la lista *Destino*, a la lista *Fuente*. El elemento seleccionado lo dejaremos caer sobre la lista *Fuente*, sólo si el control arrastrado (parámetro *Source*) es la lista *Destino*. Cuando se trate de otro control, por ejemplo de la propia lista *Fuente*, no se tendrá en cuenta. La operación mover implica añadir el elemento seleccionado de la lista *Destino* a la lista *Fuente* y borrarlo de la lista *Destino*. También, si no hay un elemento seleccionado en la lista *Fuente* (**ListIndex** = –1), se selecciona el elemento 0 de la misma.


```
Sub Fuente_DragDrop (Source As Control, X As Single, Y
As Single)
  If Source.Tag <> "Fuente" Then
    Fuente.AddItem Source.List(Source.ListIndex)
    'Si no hay un elemento seleccionado
    If Fuente.ListIndex = -1 Then
      Fuente.ListIndex = 0'seleccionar primer elemento
    End If
    BorrarElemento Source
  End If
End Sub
```

Para borrar definitivamente todos los ficheros de la lista *Destino*, el usuario hará clic en el botón *Vaciar papelera* (**Index** = 1). El código asociado con este botón realiza las siguientes operaciones: verifica si hay elementos en la lista, cambia el cursor del ratón utilizando la sentencia *Screen.MousePointer* para que indique "esperar", borra los elementos de la lista uno a uno (para borrar los ficheros correspondientes a cada elemento de la lista, incluya usted el código que le interese), modifica la apariencia de la papelera para que indique vacía y restaura el cursor del ratón. Para cerrar esta ventana, haga clic en el botón *Cerrar*.

```
Sub Orden_Click (Index As Integer)
  Dim I As Integer
  If Index = 1 Then 'vaciar la papelera
    If Destino.ListCount <= 0 Then 'si hay elementos
      Beep
      Exit Sub
    End If
    Screen.MousePointer = 11 'cambiar cursor del ratón
    For I = 1 To Destino.ListCount
      Destino.ListIndex = 0
      'Aquí escriba el código necesario para borrar
      'los elementos referenciados en la lista
      Destino.RemoveItem 0
    Next I
    Form1.Papelera.Picture = Form1.Vacía.Picture
    Screen.MousePointer = 0 'volver al cursor inicial
  Else 'cerrar la ventana
    Orden(0).SetFocus
```

```
    Form2.Hide
  End If
End Sub
```

Para borrar un elemento de la lista *Fuente* o de la lista *Destino*, escriba el siguiente módulo y guárdelo con el nombre *papemod1.bas*. En el procedimiento siguiente distinguimos dos casos: borrar un elemento que no es el primero y borrar el elemento primero. Para borrar un elemento que no es el primero seleccionamos el elemento anterior a él y después lo borramos y si se trata del primer elemento, primero lo borramos y a continuación, si quedan elementos en la lista, seleccionamos el primero. Cuando se trate de la lista *Destino* y se borre el último elemento que quede, hay también que modificar la apariencia de la papelera para que indique vacía.

```
Sub BorrarElemento (Source As Control)
  If Source.ListIndex > 0 Then
    'Borrar un elemento que no es el primero
    Source.ListIndex = Source.ListIndex - 1
    Source.RemoveItem Source.ListIndex + 1
  Else
    'Borrar primer elemento
    Source.RemoveItem Source.ListIndex
    If Source.ListCount > 0 Then 'si lista no vacía
      'seleccionar el primer elemento
      Source.ListIndex = 0
    End If
  End If
  If Source.Tag = "Destino" Then
    If Source.ListCount = 0 Then
      Form1.Papelera.Picture = Form1.Vacía.Picture
    End If
  End If
End Sub
```

CAPÍTULO 11

DEPURAR UNA APLICACIÓN

INTRODUCCIÓN

Visual Basic provee utilidades que permiten analizar como se desarrolla la ejecución desde un lugar de la aplicación a otro. Esto es una forma de determinar qué ocurre en una aplicación que aparentemente parece correcta, pero que los resultados mostrados no son satisfactorios. Las técnicas de depuración empleadas por Visual Basic incluyen, puntos de parada, ejecución paso a paso, y la posibilidad de visualizar valores de variables y de propiedades.

Cuando se realiza una aplicación se pueden presentar tres clases de errores: errores de sintaxis, errores en tiempo de ejecución y errores lógicos.

Los *errores de sintaxis* son el resultado de escribir incorrectamente una sentencia. Si tiene activada la opción **Syntax Checking** del menú **Code**, Visual Basic detecta estos errores tan pronto como se producen, esto es, en el momento de escribir una sentencia incorrecta en la ventana de código.

Los *errores en tiempo de ejecución* ocurren cuando durante la ejecución de una aplicación, una sentencia intenta una operación que es imposible realizar. Un ejemplo típico es una división por cero. Una buena programación exige que la aplicación se anticipe a muchos de estos errores, manipulándolos adecuadamente.

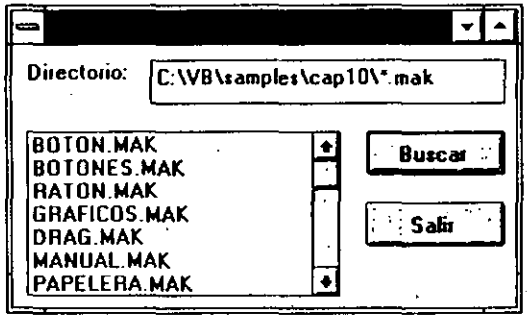
Los *errores lógicos* se producen cuando siendo la aplicación sintácticamente correcta, se ejecuta sin producirse errores en tiempo de ejecución, pero los resultados que se obtienen no son correctos. Para detectar este tipo de errores, es necesaria una minuciosa depuración, analizando cada parte de código y verificando resultados intermedios.

MANIPULACIÓN DE ERRORES

Desde el punto de vista teórico las aplicaciones no necesitan rutinas para manipulación de errores. Desde el punto de vista práctico esto no es cierto, ya que hay que prever cualquier manipulación errónea del usuario. Por ejemplo, un usuario ejecuta una aplicación para buscar determinados ficheros en la unidad A y no ha introducido el disquete. La aplicación fallará.

El siguiente ejemplo permite buscar ficheros en un directorio previamente especificado.

Esta aplicación presenta una ventana como la de la figura siguiente. En ella, la caja de texto sirve para que el usuario escriba el camino completo de los ficheros que desea buscar. Por defecto, la caja de texto muestra el directorio actual. Una vez escrito el camino de búsqueda, el usuario pulsará el botón *Buscar* y la lista de ficheros será visualizada. Si no hubiera ningún fichero de los especificados, aparecerá un mensaje indicándolo y si los hubiera, también aparecerá un mensaje indicando cuantos ficheros se han encontrado. Cuando el usuario haya realizado todas las búsquedas previstas, pulsará el botón *Salir* para finalizar la aplicación.



Inicie una nueva aplicación y cree una forma titulada *Buscar Fichero* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *dir.mak* y la forma con el nombre *dir.frm*.

Objeto	Propiedad	Valor
Forma	FormName Caption	Form1 Buscar Ficheros

Etiqueta	CtlName Caption	Label1 Directorio:
Caja de texto	CtlName TabIndex Text	Directorio 0 (nada)
Botón	CtlName Caption Default	Buscar Buscar True
Botón	CtlName Caption	Salir Salir

La propiedad **TabIndex** devuelve o asigna la posición Tab para un control dentro del conjunto de controles de una forma. Cuando usted pulsa la tecla **Tab** para dirigirse de un control a otro, el orden que se sigue es el asignado por el valor de esta propiedad. Visual Basic asigna un valor por defecto a esta propiedad. El valor de esta propiedad para el primer control dibujado es cero y cada control que se añade toma el siguiente valor. Si usted modifica uno de estos valores, Visual Basic realiza una renumeración automática para el resto de los controles.

Para asegurarnos de que al arrancar la aplicación el cursor se sitúa en la caja de texto, hemos asignado el valor cero a su propiedad **TabIndex**. Si ha pensado utilizar el método **SetFocus** en el procedimiento *Form_Load* para realizar esta operación, no lo haga ya que obtendrá un error. Una vez que la forma esté presente podrá utilizar este método.

A continuación escribimos el código para que la aplicación ejecute lo enunciado. En primer lugar escriba el código asociado al botón *Salir* para el suceso **Click**. El objetivo de este procedimiento es finalizar la aplicación.

```
Sub Salir_Click ()
  End
End Sub
```

Para que inicialmente, cuando se arranque la aplicación, aparezca en la caja de texto el directorio actual, escriba el siguiente procedimiento.

```
Sub Form_Load ()
  Form1.Directorio.Text = CurDir$
End Sub
```

La función **CurDir$** devuelve el camino actual para la unidad de disco especificada. Su sintaxis es,

**CurDir$** [*(unidad$)*]

Si la unidad no se especifica o es una cadena vacía, **CurDir$** devuelve el camino para la unidad de disco actualmente seleccionada.

Cuando el usuario haya escrito el camino de búsqueda en la caja de texto, pulsará el botón *Buscar*. Esto hará que se ejecute el procedimiento *Buscar_Click* que invocará al procedimiento que realiza el proceso de búsqueda.

```
Sub Buscar_Click ()
  BuscarFichero
End Sub
```

El procedimiento *BuscarFichero* utiliza la función **Dir$** para buscar uno o más ficheros. Su sintaxis es.

Dir$ [*(ficheroS)*]

El argumento *ficheroS* especifica el patrón del fichero o ficheros que queremos buscar. La primera vez que se llame a esta función, hay que especificar dicho argumento.

La primera vez que se ejecuta la función **Dir$**, devuelve el nombre del primer fichero que coincida con las especificaciones realizadas. Para seguir buscando más ficheros con las misma especificaciones, ejecute **Dir$** sin argumentos. Cuando la cadena devuelta sea nula, quiere decir que la búsqueda ha finalizado.

En nuestra aplicación, los ficheros encontrados por la función **Dir$** son añadidos a una lista y cuando la búsqueda finaliza se enfoca de nuevo la caja de texto. En el caso de realizar más de una búsqueda, cada lista quedaría añadida a la anterior; para que esto no suceda, antes de formar una nueva lista borramos la anterior si existe.

Abra la ventana de código para escribir un nuevo procedimiento (**(general)**) y escriba el código que se presenta a continuación.

```
Sub BuscarFichero ()
  'Utilización de Dir$ para buscar ficheros
  '    Fichero As String, Msg As String, Conta As Integer
       INFORMACIÓN = 64 'Iconos para MsgBox
```



```
  'Si ya existe una lista borrarla
  If Form1.List1.ListCount Then
    For I = 1 To Form1.List1.ListCount - 1
      Form1.List1.RemoveItem 0
    Next I
  End If
  'Visualizar la lista de ficheros buscados
  Fichero = Dir$(Form1.Directorio.Text) 'busca el primero
  Do While Len(Fichero)
    Form1.List1.AddItem Fichero
    Conta = Conta - 1
    Fichero = Dir$ 'sin argumentos busca el siguiente
  Loop

  If Form1.List1.ListCount Then
    Msg = "Se encontraron" + Str$(Conta) + " ficheros"
  Else
    Msg = "No se encontraron ficheros"
  End If
  MsgBox Msg, INFORMACIÓN, Form1.Caption
  Form1.Directorio.SetFocus
End Sub
```

El procedimiento que acabamos de escribir, teóricamente funciona correctamente; incluso, lo hemos probado con nuestro disco duro y no hemos detectado ningún fallo. Sin embargo, si se especifica una unidad de disquete y ésta no tiene dentro el disquete o no tiene la puerta cerrada, el procedimiento falla interrumpiéndose la ejecución.

Para evitar esta situación utilice rutinas para manipulación de errores. Visual Basic tiene la sentencia **On Error** que permite interceptar cualquier error que ocurra durante la ejecución para así poder manipularlo. Esta sentencia y la sentencia **Resume** fueron explicadas en el Capítulo 8, en el apartado "Control de errores". Por ejemplo, los problemas expuestos anteriormente pueden ser manipulados con el siguiente código.

```
Sub BuscarFichero ()
  'Utilización de Dir$ para buscar ficheros
  Dim Fichero As String, Msg As String
  Dim Conta As Integer, I As Integer

  'Errores
  'Unidad de disco no lista = 71
  'Dispositivo no disponible = 68

  'Iconos para MsgBox
  Const INFORMACIÓN = 64, AVISO = 48, CRITI
```

```
'Activar la interceptación de errores
On Error GoTo RutinaDeError

'Si ya existe una lista borrarla
If Form1.List1.ListCount Then
  For I = 0 To Form1.List1.ListCount - 1
    Form1.List1.RemoveItem 0
  Next I
End If
'Visualizar la lista de ficheros buscados
Fichero = Dir$(Form1.Directorio.Text) 'busca el primero
Do While Len(Fichero)
  Form1.List1.AddItem Fichero
  Conta = Conta + 1
  Fichero = Dir$ 'sin argumentos busca el siguiente
Loop

If Form1.List1.ListCount Then
  Msg = "Se encontraron" + Str$(Conta) + " ficheros"
Else
  Msg = "No se encontraron ficheros"
End If
MsgBox Msg, INFORMACIÓN, Form1.Caption
Form1.Directorio.SetFocus
Exit Sub

RutinaDeError:
  'If (Err = 71) Or (Err = 68) Then
    Msg = "Introducir disquete y cerrar la puerta"
    MsgBox Msg, AVISO
  Else
    Msg = "Ha ocurrido el error" + Str$(Err) + " no previsto"
    Msg = Msg + Chr$(10) + Chr$(13) + Error$
    MsgBox Msg, CRITICO
    Stop 'suspende la ejecución
  End If
  Resume
End Sub
```

La función **Err** devuelve el código del último error ocurrido durante la ejecución. La lista de errores puede verla en la ayuda de Visual Basic (solicite ayuda, pulse el botón **Search** y busque **Trappable errors**).

La función **Error$** devuelve el mensaje correspondiente al último error ocurrido. Si a continuación se especifica un número entero, **Error$** *(n)*, entonces el mensaje de error devuelto corresponde a ese código.

Si en algún caso desea desactivar la interceptación de errores, ejecute la sentencia **On Error GoTo 0**. Si desea simular que ha ocurrido un error ejecute la sentencia **Error** *código*.

Cuando ocurre un error en un procedimiento que no tiene una rutina de manipulación, o cuando ocurre un error dentro de una rutina de manipulación, Visual Basic busca hacia atrás en la ruta de invocación otra rutina de manipulación. Por ejemplo, considere que un procedimiento A que tiene una rutina de manipulación, llama a un procedimiento B y que éste a su vez llama a un procedimiento C. Mientras el procedimiento C se está ejecutando, la ruta de invocación es A, B, C. Si ocurre un error en el procedimiento C que no tiene rutina de manipulación, Visual Basic busca por ella primero en B y después en A. Cuando Visual Basic encuentra una rutina de manipulación la ejecuta; si no la encuentra, entonces visualiza el mensaje de error apropiado y detiene la ejecución.

DEPURACIÓN

Durante el diseño y puesta en marcha de una aplicación hay tres modos en los que podemos estar: diseño (**design**), ejecución (**run**), o pausa (**break**). Mire a la barra de título de Visual Basic y observe que siempre le muestra el modo en el que se encuentra.

El modo *diseño* se caracteriza porque es donde se realiza la mayor parte del trabajo de creación de la aplicación. En este modo se diseñan formas, se dibujan controles, se escribe código, y se utiliza la barra de propiedades para ver o modificar algunas de ellas, pero no se puede ejecutar el código.

El modo *ejecución* se caracteriza porque aquí la aplicación toma el control. En este modo podemos ver el código pero no podemos modificarlo. Para pasar de este modo al de *diseño*, ejecute la orden **End** del menú **Run**; para pasar al modo *pausa*, ejecute la orden **Break** del menú **Run** o también, pulse las teclas **Ctrl+Break**.

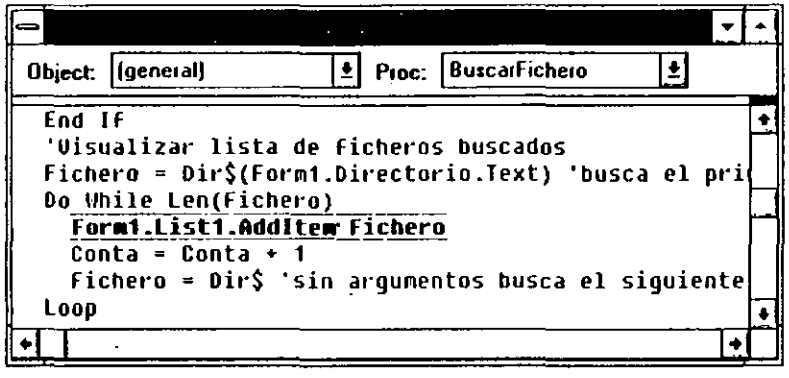
El modo *pausa* se caracteriza porque permite ejecutar la aplicación paso a paso. En este modo se puede editar código, examinar variables y propiedades, dar un paso más en la ejecución, ejecutar la aplicación hasta el final, iniciar la ejecución, o finalizar la misma.

Para entrar a depurar una aplicación, es necesario entrar en el modo *pausa*. Esto lo puede hacer de varias formas, dependiendo del lugar donde quiera entrar en la aplicación. Si quiere entrar a realizar una ejecución paso a paso justamente desde le principio pulse la tecla **F8**. Si la ejecución paso a paso la quiere reali-

zar a partir de un determinado lugar dentro de la aplicación, coloque en ese punto un **Stop** o un punto de parada. Ambos detienen la ejecución de la aplicación pasando del modo de *ejecución* al modo *pausa*.

Un punto de parada puede ponerse en tiempo de diseño o en tiempo de ejecución. Para poner o quitar un punto de parada, sitúe el cursor sobre la línea donde desea se produzca la interrupción temporal de la ejecución y pulse la tecla **F9**. Cuando en una línea hay un punto de parada ésta aparece en negrita.

Por ejemplo, cargue la aplicación *dir.mak* anterior y observe el título de la ventana de Visual Basic, pone [design]. Ponga en la aplicación anterior un punto de parada en el lugar que indica la figura siguiente. A continuación pulse **F5** para ejecutar la aplicación y observe de nuevo el título de la ventana de Visual Basic, pone [run]. Pulse el botón *Buscar* y observará que la ejecución se detiene donde ha puesto el punto de parada; ahora la ventana de Visual Basic pone [break].



Entre colocar un punto de parada o una sentencia **Stop** hay una diferencia importante. Cuando se abandona la aplicación y se carga otra vez, se observa que todos los puntos de parada fueron borrados. En cambio, las sentencias **Stop** son permanentes.

Ahora pruebe a pulsar **F8** una y otra vez. Observará que la ejecución continua paso a paso.

Si en lugar de pulsar **F8** pulsa **Shift+F8** el proceso es idéntico, excepto cuando la sentencia es una llamada a un procedimiento, como ocurre en el procedimiento *Buscar_Click*. En este caso la llamada se ejecuta en un solo paso, esto es, el procedimiento *Buscar_Click* no se ejecuta paso a paso.

Como ejemplo, detenga la ejecución, si aún no lo ha hecho, y pruebe desde el inicio las dos modalidades de ejecución anteriores, **F8** y **Shift+F8**.

Otras veces, iniciada la ejecución paso a paso, puede querer iniciar la ejecución desde una línea anterior, esto es, desde una línea por la que ya ha pasado. Quizá desee hacer esto porque desde la ventana inmediata (**Immediate Window**) ha cambiado el valor de alguna variable o propiedad y quiere ver los resultados que se producen. Visual Basic le permite realizar esta operación desde diferentes puntos dentro del mismo procedimiento. Para ello, coloque el cursor en la línea donde desee iniciar la ejecución y ejecute la orden **Set Next Statement** del menú **Run**.

Si quiere reanudar la ejecución hasta el final, pulse la tecla **F5**.

Ventana inmediata

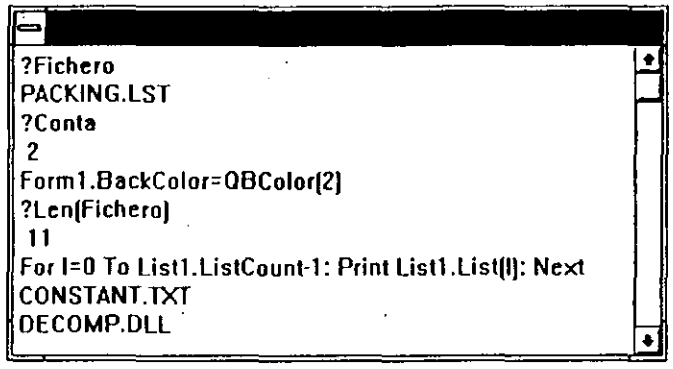
La mayoría de las veces, para encontrar la causa de un problema mientras ejecutamos paso a paso una parte del código de nuestra aplicación, necesitamos asignar valores, evaluar expresiones y verificar procedimientos. Este tipo de operaciones son las que nos permite realizar la ventana inmediata (**Immediate Window**).

Para visualizar valores en esta ventana hay dos formas: utilizar la sentencia **Print** o **?** directamente sobre ésta ventana para escribir los valores actuales de las variables y propiedades deseadas, o colocar sentencias **Debug.Print** en el código de la aplicación.

Una vez que ha entrado en la ventana inmediata, haciendo clic sobre ella o ejecutando la orden **Immediate Window** del menú **Window**, para moverse dentro ella utilice las teclas convencionales de cualquier procesador de textos (teclas de movimiento del cursor, página arriba, página abajo, suprimir, etc.).

Por ejemplo, coloque un punto de parada en el procedimiento *BuscarFichero* en el lugar que se indicó en la figura anterior. A continuación pulse **F5** para ejecutar la aplicación, escriba un camino en la ventana de texto y pulse el botón *Buscar*. Observará que la ejecución se detiene donde ha puesto el punto de parada. Active la ventana inmediata haciendo clic sobre ella y pregunte por los valores de las variables y propiedades que desee, como se indica en la figura siguiente.

También es posible asignar valores, evaluar expresiones y verific[illegible]ultados de procedimientos.



La otra forma de imprimir datos en la ventana inmediata era escribiendo en los lugares apropiados del procedimiento sentencias de la forma.

```
Debug.Print Fichero
Debug.Print Form1.BackColor
```

También, si su aplicación está preparada para admitir argumentos en la línea de órdenes para cuando produzca un fichero ejecutable, para probar desde Visual Basic como se comporta con estos argumentos, tiene en el menú **Run** la orden **Modify Command$...** que le permite en tiempo de diseño, entrar dichos argumentos.

La función **Command$** de Visual Basic, precisamente devuelve como resultado los argumentos enviados. Si esta cadena es nula, no hay argumentos.

CONSIDERACIONES ESPECIALES

Cuando se depura un programa es importante pensar en las dificultades que nos pueden presentar procedimientos asociados a algunos sucesos, como los referentes al ratón o al teclado.

Por ejemplo, si hace una pausa durante la ejecución de un procedimiento asociado con el suceso **MouseDown**, suelta el botón del ratón, realiza las operaciones que estima convenientes, etc., cuando continúe la ejecución, se le presentará el problema de que la aplicación asume que el botón del ratón está todavía pulsado. Quiere esto decir, que no se reconocerá un suceso **MouseUp** hasta que no pulse otra vez el botón del ratón y lo suelte.

Mismo efecto ocurre con los sucesos **KeyDown** y **KeyUp**.

PARTE 2

Técnicas avanzadas

- Ficheros indexados
- Intercambio dinámico de datos
- Llamadas a las funciones de la API de Windows
- Comunicaciones

CAPÍTULO 12

FICHEROS INDEXADOS

INTRODUCCIÓN

Existen tres organizaciones de ficheros básicas, de cuya combinación se derivan multitud de organizaciones posibles. Estas son:

- Secuencial
- Aleatoria
- Secuencial indexada

En cada caso, se elegirá una u otra en función de las características de los soportes y del modo de acceso requerido.

En cuanto a los tipos de acceso, distinguimos dos:

- Acceso secuencial
- Acceso aleatorio o directo

Se habla de acceso secuencial cuando se van accediendo posiciones sucesivas, esto es, tras acceder a la posición N se accede a la posición N+1; y se habla de acceso directo cuando se accede directamente a la posición deseada, sin necesidad de acceder a las posiciones anteriores.

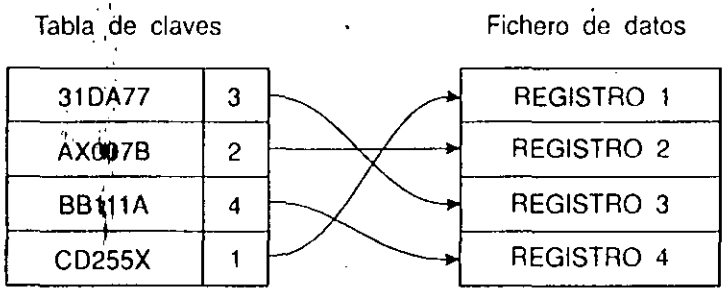
En el Capítulo 8 ya se expusieron los ficheros secuenciales y los ficheros aleatorios. A continuación veremos como trabajar con ficheros indexados. Algunos lenguajes, como por ejemplo COBOL, incorporan este tipo de ficheros. Visual Basic, lo mismo que Basic, QBasic o QuickBasic, no incluyen este tipo de ficheros por lo que tiene que ser el usuario el que se los construya. Otra solución sería adquirir un paquete estándar como, por ejemplo, VB ISAM o BTRIEVE.

ESTRUCTURA DE UN FICHERO SECUENCIAL INDEXADO

La organización secuencial indexada es un modelo de almacenamiento de datos que se apoya en una tabla de claves y en un fichero de datos. La tabla de claves actúa como índice del fichero de datos. Un ejemplo típico de este tipo de almacenamiento es un libro, ya que éste consta de un índice ordenado por número de página y el contenido del libro en sí. De esta forma, para acceder a una determinada información, buscaremos en el índice el número de página donde se encuentra, y a continuación iremos directamente a esa página.

Las claves de acceso que contiene la tabla, generalmente está clasificada en orden ascendente y cada una de ellas se corresponde con un registro del fichero de datos. Una clave puede ser un *código de artículo*, un *dni*, etc.

Ahora bien, para hacer corresponder una clave de la tabla con un registro del fichero de datos, se precisa convertir ésta en un número que nos de su posición dentro de la tabla y nos permita así acceder al número que tiene el registro dentro del fichero de datos. Existen diferentes algoritmos para realizar este proceso. Entre ellos, los más utilizados y eficientes son los *algoritmos Hash*. También es verdad, que la organización secuencial indexada es un modelo de almacenamiento que no precisa de esta conversión de claves, ya que cada una de ellas puede directamente emparejarse con el primer registro libre del fichero de datos. De hacerlo así, en la tabla de claves se almacenarán las parejas clave, número de registro. Esta forma de proceder es menos eficiente que la anteriormente citada.



Según lo expuesto, para crear un fichero secuencial indexado se tendrán que crear dos ficheros, uno con las claves denominado *fichero índice*, y otro con los datos denominado *fichero de datos*. Para acceder a un determinado registro del fichero secuencial indexado, primero se busca la clave en el fichero índice, lo cual nos proporcionará el número del registro que a continuación leeremos en el fichero de datos.

La clave de acceso no tiene por qué ser única, esto es, para acceder a un fichero podemos utilizar si es preciso varias claves de acceso, lo cual requerirá otras tantas tablas índice. Dicho de otra forma, el nivel de indexación puede ser también múltiple, lo que quiere decir que la búsqueda en el índice se efectúe por medio de otro índice (índices jerarquizados o en árbol).

ALGORITMOS HASH

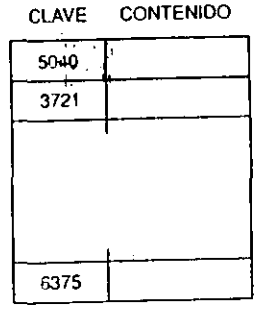
Los algoritmos **Hash** son métodos de búsqueda, que proporcionan una "**longitud de búsqueda**" pequeña y una flexibilidad superior a la de otros métodos, como puede ser, el método de "**búsqueda binaria**" que requiere que los elementos de la tabla estén ordenados.

Por "**longitud de búsqueda**" se entiende el número de accesos que es necesario efectuar sobre una tabla para encontrar el elemento deseado.

Este método de búsqueda permite, como operaciones básicas, además de la búsqueda de un elemento, insertar un nuevo elemento (si la tabla está vacía, crearla) y eliminar un elemento existente.

Tablas Hash

Una tabla producto de la aplicación de un algoritmo **Hash** se denomina "**tabla Hash**" y son las tablas que se utilizan con mayor frecuencia en los sistemas de acceso. Gráficamente estas tablas tienen la siguiente forma:



La tabla se organiza con elementos formados por dos miembros: *clave* y *contenido*.

La *clave* constituye el medio de acceso a la tabla. Aplicando a la *clave* una función de acceso "*fa*", previamente definida, obtenemos un número entero positivo "*i*", que nos da la posición del elemento correspondiente, dentro de la tabla.

$$i = fa(clave)$$

Conociendo la posición, tenemos acceso al *contenido*. El miembro *contenido* puede albergar directamente la información, o bien un puntero a dicha información, cuando ésta sea muy extensa.

El acceso, tal cual lo hemos definido, recibe el nombre de "**acceso directo**"

Como ejemplo, suponga que la *clave* de acceso se corresponde con el número del documento nacional de identidad (*dni*) y que el contenido son los datos correspondientes a la persona que tiene ese *dni*. Una función de acceso, *i=fa(dni)*, que haga corresponder la posición del elemento en la tabla con el *dni*, es inmediata:

$$i = dni$$

la cual da lugar a un acceso directo. Esta función así definida presenta un inconveniente y es que el número de valores posibles de "*i*" es demasiado grande para utilizar una tabla de este tipo.

Para solucionar este problema, es posible siempre, dada una tabla de longitud *L*, crear una "**función de acceso**", *fa*, de forma que genere un valor comprendido entre *0* y *L*, más comúnmente entre *1* y *L*.

En este caso puede suceder que dos o más claves den un mismo valor de "*i*":

$$i = fa(clave_1) = fa(clave_2)$$

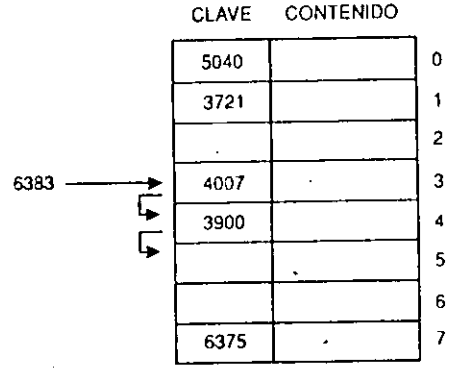
El método **Hash** está basado en esta técnica: el acceso a la tabla es directo por el número "*i*" y cuando se produce una **colisión** (dos claves diferentes dan un mismo número "*i*") este elemento se busca en una zona denominada "**área de overflow**".

Método Hash abierto

Este es uno de los métodos más utilizados. El algoritmo para acceder a un elemento de la tabla de longitud *L*, es el siguiente:

1. Se calcula *i = fa(clave)*.

2. Si la posición "*i*" de la tabla está libre, se inserta la *clave* y el *contenido*. Si no está libre y la *clave* es la misma, error: clave duplicada. Si no está libre y la *clave* es diferente, incrementamos "*i*" en una unidad y repetimos el proceso descrito en este punto.



En la figura observamos que queremos insertar la clave 6383. Supongamos que aplicando la función de acceso obtenemos un valor de 3, esto es,

$$i = fa(6383) = 3$$

Como la posición 3 está ocupada y la clave es diferente, tenemos que incrementar "*i*" y volver de nuevo al punto 2 del algoritmo.

La "**longitud media de búsqueda**" en una "**tabla Hash abierta**" viene dada por la expresión:

$$accesos = (2-k)/(2-2k)$$

siendo *k* igual al número de elementos existentes en la tabla dividido por *L*.

Ejemplo:

Si existen *60* elementos en una tabla de longitud *L=100*, el número medio de accesos para localizar un elemento será:

$$accesos = (2-60/100)/(2-2*60/100) = 1,75$$

En el método de "búsqueda binaria", el número de accesos viene dado por $\log_2 N$, siendo N el número de elementos de la tabla.

Para reducir al máximo el número de colisiones y, como consecuencia, obtener una "**longitud media de búsqueda**" baja, es importante elegir bien la función de acceso.

Una "**función de acceso**" o "**función Hash**" bastante utilizada y que proporciona una distribución de las claves uniforme y aleatoria es la "**función mitad del cuadrado**" que dice: "dada una clave *C*, se eleva al cuadrado (C^2) y se cogen *n* bits del medio, siendo $2^n <= L$.

Ejemplo:

Supongamos: L = 256 lo que implica n = 8

C = 625

$C^2 = 390625$ ($0 <= C^2 <= 2^{32}-1$)

$390625_{10} = 00000000000001011111010111100001_2$

n bits del medio: $01011111_2 = 95_{10}$

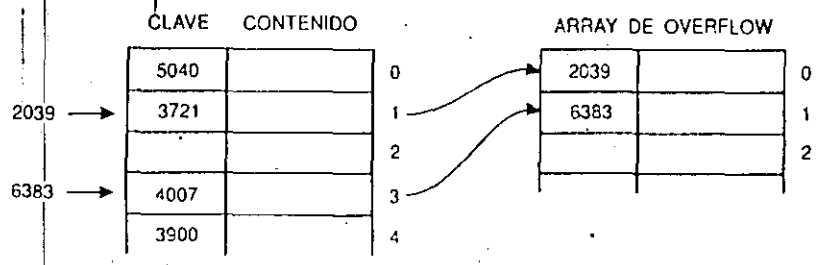
Otra "**función de acceso**" muy utilizada es la "**función módulo**" (resto de una división entera):

i = *clave* **módulo** *L*

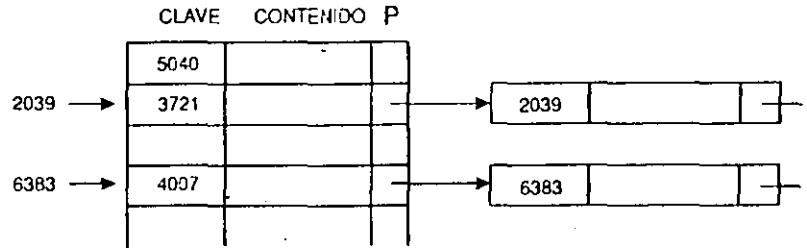
Cuando se utilice esta función es importante elegir un número primo para *L*, con la finalidad de que el número de colisiones sea pequeño.

Método Hash con overflow

Una alternativa al método anterior es la de disponer de otra tabla separada, para insertar las claves que producen colisión, denominada "**tabla de overflow**", en la cual se almacenan todas estas claves de forma consecutiva.



Otra forma alternativa más normal es organizar una lista encadenada por cada posición de la tabla donde se produzca una colisión.



Cada elemento de esta estructura incorpora un nuevo miembro *P*, el cual referencia la lista encadenada de overflow.

Eliminación de elementos

En el método **Hash** la eliminación de un elemento no es tan simple como dejar vacío dicho elemento, ya que esto da lugar a que los elementos insertados por colisión no puedan ser accedidos. Por ello se suele utilizar un miembro (campo) complementario que sirva para poner una marca de que dicho elemento está eliminado. Esto permite acceder a otros elementos que dependen de él por colisiones, ya que la clave se conserva y también permite insertar un nuevo elemento en esa posición cuando se dé una nueva colisión.

PROCESO DE UN FICHERO SECUENCIAL INDEXADO

Para crear un fichero secuencial indexado, los pasos a seguir son los siguientes:

1. Crear el fichero índice.
2. Crear el fichero de datos.

El acceso a los datos de un fichero secuencial indexado debe hacerse siempre en dos pasos:

1. Acceso al fichero índice para buscar la clave.
2. Acceso directo al fichero de datos.

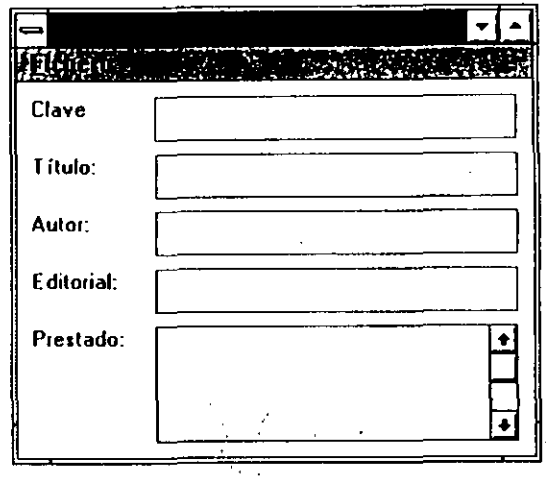
Por ejemplo, crear un fichero indexado que contenga los datos de cada uno de los libros de nuestra biblioteca particular. Por cada uno de los libros vamos a almacenar: una clave que lo identifique, título del libro, autor, editorial, datos sobre el préstamo y un campo, borrado, que nos permita identificar un registro como eli-

minado, pero sin borrarlo físicamente. Esto permite una velocidad mayor de ejecución y posibilita en un momento determinado obtener un fichero con los libros dados de baja, actualizar el fichero maestro y reconstruir el índice.

Inicie una nueva aplicación y cree una forma titulada *Biblioteca* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *libros.mak* y la forma con el nombre *libros.frm*.

Objeto	Propiedad	Valor
Menú Fichero	Caption CtlName	&Fichero MenúFichero
Fichero: orden 0	Caption CtlName Accelerator	Ab&rir... FicheroAbrir Ctrl+R
Fichero: orden 1	Caption CtlName Accelerator	&Guardar como... FicheroGuardar Ctrl+G
Fichero: orden 2	Caption CtlName Accelerator	&Añadir Registro AñadirReg Ctrl+A
Fichero: orden 3	Caption CtlName Accelerator	&Buscar Registro BuscarReg Ctrl+B
Fichero: orden 4	Caption CtlName Accelerator	B&orrar Registro BorrarReg Ctrl+O
Fichero: orden 5	Caption CtlName Accelerator	&Salir Salir Ctrl+S
Etiqueta 1	Caption CtlName	Clave: Label1
Caja de texto 1	CtlName Text	Clave (nada)
Etiqueta 2	Caption CtlName	Título: Label2
Caja de texto 2	CtlName Text	Título (nada)
Etiqueta 3	Caption CtlName	Autor: Label3
Caja de texto 3	CtlName Text	Autor (nada)

Etiqueta 4	Caption CtlName	Editorial: Label4
Caja de texto 4	CtlName Text	Editorial (nada)
Etiqueta 5	Caption CtlName	Prestado: Label5
Caja de texto 5	CtlName Text MultiLine ScrollBars	Prestado (nada) True 2 - Vertical



Cada registro del fichero de datos contendrá los campos, *Clave*, *Título*, *Autor*, *Editorial*, *Prestado* y *Borrado*. El campo *Borrado* inicialmente tendrá un valor 0 y cuando se marque el registro como borrado, tendrá un valor –1.

Cada registro del fichero índice contendrá dos campos: *Clave*, la misma que figura en el fichero de datos y *NumReg*, número de registro del fichero de datos, donde se guardan los datos correspondientes al libro identificado con esa clave.

Añada las siguientes declaraciones al módulo global y denomínelo *libros.bas*.

```
Global Const TRUE = -1
Global Const FALSE = 0

Type Registro
  Clave As Long
  Título As String * 30
```

```
  Autor As String * 30
  Editorial As String * 12
  Prestado As String * 240
  Borrado As Integer '0=no borrado, -1=borrado
End Type
Global Libros As Registro
Global FicheroD As String 'fichero de datos

Type TipoRegInd
  Clave As Long
  NumReg As Integer
End Type
Global RegInd As TipoRegInd
Global L As Integer 'longitud del fichero índice
Global FicheroI As String 'fichero índice
```

El hecho de almacenar la clave en el fichero de datos cuando sería suficiente almacenarla solamente en el fichero índice, tiene ventajas como la de reconstruir el fichero índice con muy poco esfuerzo, una vez se hayan eliminado físicamente los registros marcados como borrados.

Cuando el usuario desee crear un fichero nuevo, primero introducirá los datos correspondientes a un registro, después ejecutará la orden *Añadir Registro* del menú *Fichero* y así sucesivamente. Por último ejecutará la orden *Guardar como* para asignar un nombre al fichero. Esto exige inicializar las variables *FicheroD* y *FicheroI* a un valor por defecto, operación que realiza el procedimiento *Form_Load*.

```
Sub Form_Load ()
  On Error GoTo RutinaError
  Close
  Kill "Temp.dat"
  Kill "Temp.ind"
  FicheroD = "Temp.dat"
  FicheroI = "Temp.ind"
  CrearÍndice 'genera un fichero índice vacío
  Exit Sub
RutinaError:
  Resume Next
End Sub
```

La sentencia *Kill* asegura que inicialmente partamos de una situación totalmente nueva, borrando, si existen, los ficheros de datos e índice utilizados por defecto y denominados *Temp.dat* y *Temp.ind* respectivamente. Si el fichero especificado para borrar no existe, se ejecuta la sentencia *Resume Next*.

El procedimiento *CrearÍndice* inicializa a 0, los campos clave y número de registro de cada uno de los registros del fichero índice. También calcula el número de registros del fichero índice. Por ejemplo, si deseamos tener 1000 registros útiles y una longitud media de búsqueda de 3, aplicando la teoría relativa al método **Hash abierto**, el fichero índice debe tener *L=1000/0.8* registros.

Inicie un nuevo módulo llamado *índice.bas* (orden **New Module** del menú **File**) y añada al mismo el siguiente procedimiento.

```
Sub CrearÍndice ()
  Dim I As Integer
  'Crear un fichero índice temp.ind con
  'L registros inicialmente vacíos, de los
  'que se prevé sean ocupados el 80%=1000
  L = 1000 / .8 'longitud media de búsqueda = 3
  L = Siguiente_Primo(L)
  Open FicheroI For Random As #1 Len = Len(RegInd)
  For I = 1 To L
    RegInd.Clave = 0
    RegInd.NumReg = 0
    Put #1, I, RegInd
  Next I
  Close #1
End Sub
```

Por otra parte, vimos que era importante elegir un número primo para *L*, con la finalidad de que el número de colisiones sea pequeño. De esto se encarga la función *Siguiente_Primo*. Añada este procedimiento al módulo *índice.bas*.

```
Function Siguiente_Primo (N As Integer) As Integer
  Dim Primo As Integer, I As Integer
  Primo = FALSE
  If (N Mod 2 = 0) Then N = N + 1
  Do While Not Primo
    Primo = TRUE
    For I = 3 To Sqr(N) Step 2
      If (N Mod I = 0) Then Primo = FALSE
    Next I
    If (Not Primo) Then N = N + 2
  Loop
  Siguiente_Primo = N
End Function
```

Una vez creado el fichero índice, pasamos a crear el fichero de datos. El acceso a los datos de un fichero secuencial indexado debe hacerse siempre en dos pasos:

1. Se accede al fichero índice para buscar la clave. lo cual. también proporciona la posición del registro correspondiente en del fichero de datos.
2. Se accede directamente a ese registro del fichero de datos.

Acceso al fichero índice

Cuando se accede al fichero índice puede ser por una de las dos causas siguientes:

- porque hay que añadir un nueva clave,
- o porque hay que buscar una determinada clave.

Para añadir una nueva clave invocaremos a la función *HashIn* que se describe a continuación. Esta función devuelve un 1 si se añadió la clave. un 2 si la clave ya existe, y un 3 si el fichero índice está lleno. El desarrollo de la misma, se ha hecho aplicando los conceptos teóricos explicados anteriormente para el método **Hash abierto**. Añada este procedimiento al módulo *índice.bas*.

```
Function HashIn (RegInd As TipoRegInd) As Integer
  Dim I As Long, Conta As Integer, Insertado As Integer
  Dim Reg As TipoRegInd 'registro actual
  Conta = 1 'contador
  Insertado = FALSE
  Open FicheroI For Random As #2 Len = Len(RegInd)
  I = RegInd.Clave Mod L 'función de acceso
  Do While (Not Insertado And Conta < L)
    Get #2, I, Reg
    If (Reg.Clave = 0) Then 'elemento libre
      Put #2, I, RegInd
      Insertado = TRUE
      HashIn = 1 'se insertó la clave
    ElseIf (RegInd.Clave = Reg.Clave) Then 'clave duplicada
      MsgBox "Error clave duplicada", 48, "Libros"
      Insertado = TRUE
      HashIn = 2 'clave duplicada
    Else 'colisión
      I = I + 1 'siguiente registro
      Conta = Conta + 1
      If (I = L) Then I = 1
    End If
  Loop
  Close #2
  If (Conta = L) Then
    MsgBox "Error fichero índice lleno", 48, "Libros"
    HashIn = 3
  End If
End Function
```


Puesto que *RegInd* se ha definido en el módulo global, no sería necesario que esta función recibiera este valor como parámetro. Simplemente se ha especificado para dar una visión más clara de lo que se está haciendo.

Para buscar una clave invocaremos a la función *HashOut* que se describe a continuación. Esta función devuelve el número de registro del fichero de datos correspondiente a esa clave, o un 0 si la clave no existe. El desarrollo de la misma, se ha hecho aplicando los conceptos teóricos explicados anteriormente para el método **Hash abierto**. Añada este procedimiento al módulo *índice.bas*.

```
Function HashOut (Clave As Long) As Integer
  Dim I As Long, Conta As Integer, Encontrada As Integer
  Dim Reg As TipoRegInd 'registro actual
  If Clave = 0 Then Exit Function
  Conta = 1 'contador
  Encontrada = FALSE
  Screen.MousePointer = 11 'reloj de arena
  Open FicheroI For Random As #2 Len = Len(RegInd)
  I = Clave Mod L 'función de acceso
  Do While (Not Encontrada And Conta < L)
    Get #2, I, Reg
    If (Reg.Clave = Clave) Then 'clave encontrada
      HashOut = Reg.NumReg
      Encontrada = TRUE
    Else 'o no está o está más adelante
      I = I + 1 'siguiente registro
      Conta = Conta + 1
      If (I = L) Then I = 1
    End If
  Loop
  Close #2
  If (Conta = L) Then
    MsgBox "Error: no existe un libro con esa clave", 48, "Libros"
    HashOut = 0
  End If
  Screen.MousePointer = 0 'restaurar puntero
End Function
```

Observe que para aplicar la función de acceso **Mod** necesitamos utilizar un valor numérico. Esto no quiere decir que la clave tenga que ser numérica, como sucede en nuestro ejemplo, sino que puede ser alfanumérica. Cuando se trabaje con claves alfanuméricas o alfabéticas, por ejemplo *Título*, antes de aplicar la función de acceso es necesario convertir dicha clave en un valor numérico, utilizando un algoritmo adecuado.

Para utilizar estas funciones en otras aplicaciones, tiene que cargar el módulo *indice.bas* utilizando la orden **Add File** del menú **File** y definir en el módulo global el registro *RegInd*.

Supongamos ahora que el usuario va a crear un fichero nuevo. Una vez arrancada la aplicación, introducirá los datos correspondientes a un registro, a continuación ejecutará la orden *Añadir Registro* del menú *Fichero* y así sucesivamente hasta finalizar la entrada. Para realizar el proceso de añadir un registro, abra la ventana de código correspondiente a esta orden y escriba el siguiente procedimiento.

```
Sub AñadirReg_Click ()
  Dim NumRegs As Integer, vr As Integer
  RegInd.Clave = Val(Form1.Clave.Text)
  Libros.Clave = RegInd.Clave
  Libros.Título = Form1.Título.Text
  Libros.Autor = Form1.Autor.Text
  Libros.Editorial = Form1.Editorial.Text
  Libros.Prestado = Form1.Prestado.Text
  Libros.Borrado = 0
  Open FicheroD For Random As #1 Len = Len(Libros)
  NumRegs = LOF(1) \ Len(Libros)
  RegInd.NumReg = NumRegs - 1
  vr = HashIn(RegInd)
  If vr = 3 Then End 'fichero índice lleno
  If vr = 1 Then
    Put #1, NumRegs - 1, Libros
  End If
  Close #1
End Sub
```

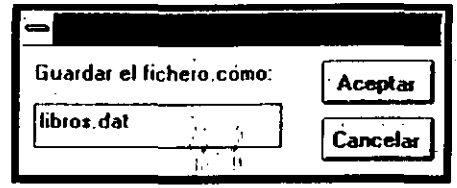
Cada nuevo registro introducido es añadido al final del fichero de datos. Para ello, este procedimiento acepta los datos del nuevo registro, calcula el número total de registros que tiene dicho fichero, *NumRegs*, y llama a la función *HashIn* para añadir al fichero índice, si es posible, la clave y la posición, *NumRegs+1*, que el registro ocupará en el fichero de datos.

Para obligar a que la clave sea un valor numérico, abra la ventana de código correspondiente a la caja de texto *Clave* y escriba el procedimiento *Clave_KeyPress*, asociado con la caja de texto *Clave* y conducido por el suceso **KeyPress**, como se indica a continuación.

```
Sub Clave_KeyPress (KeyAscii As Integer)
  Dim Car As String * 1
  Car = Chr$(KeyAscii)
  If (Car < "0" Or Car > "9") And Car <> Chr$(8) Then
    Beep          'pitido
    KeyAscii = 0  'borrar carácter
  End If
End Sub
```


Finalizado el proceso de entrada de datos, tenemos que guardar el fichero. Para realizar esta operación cree una nueva forma titulada *Guardar Fichero* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *libros.mak* y la forma con el nombre *guardarL.frm*.

Objeto	Propiedad	Valor
Forma	Caption FormName	Guardar Fichero GuardarF
Etiqueta	Caption CtlName	Guardar el fichero como: Label1
Caja de texto	CtlName Text	NombreFg (nada)
Botón 1	Caption CtlName	Aceptar Aceptar
Botón 2	Caption CtlName	Cancelar Cancelar



Cuando el usuario pulse la orden *Guardar como* del menú *Fichero* se visualizará esta forma. Abra la ventana de código correspondiente a la orden *Guardar como* del menú *Fichero* y escriba el siguiente código.

```
Sub FicheroGuardar_Click ()
  GuardarF.Show
  GuardarF.NombreFg.Text = FicheroD
End Sub
```

Este procedimiento visualiza la forma y pone en la caja de texto el nombre actual del fichero con el que estamos trabajando. Si es nuevo aparecerá *Temp.dat*.

A continuación, el usuario escribirá el nombre del fichero en la caja de texto (si es necesario, puede especificar el camino completo) y pulsará el botón *Aceptar* para guardar el fichero con el nombre especificado o *Cancelar* para anular la operación.

Abra la ventana de código correspondiente a la orden *Aceptar* y escriba el código que se muestra a continuación.

```
Sub Aceptar_Click ()
  Dim I As Integer, Cadena As String, NombreI As String
  On Error GoTo RutinaDeError
  Name FicheroD As "Temp._d_"
  Name FicheroI As "Temp._i_"
  'Componer el nombre del fichero índice
  I = InStr(Len(NombreFg.Text) - 4, NombreFg.Text, ".")
  If I <> 0 Then
    NombreI = LeftS(NombreFg.Text, I) + "ind"
  Else
    NombreI = NombreFg.Text + ".ind"
  End If
  'Verificar si el fichero de datos existe
  Open NombreFg.Text For Random As #1 Len = Len(Libros)
  If (LOF(1) <> 0) Then
    Cadena = "El fichero ya existe" - ChrS(13) + ChrS(10)
    Cadena = Cadena + "¿desea sobrescribirlo?"
    If MsgBox(Cadena, 36, "Libros") = 6 Then
      Kill NombreI 'borrar fichero índice correspondiente
    Else
      Close #1
      NombreFg.SetFocus
      Exit Sub
    End If
  End If

  Close #1
  Kill NombreFg.Text 'borrar fichero de datos

  'Nuevo nombre para FicheroD
  Name "Temp._d_" As NombreFg.Text
  FicheroD = NombreFg.Text 'nuevo fichero de datos creado

  'Nuevo nombre para FicheroI
  Name "Temp._i_" As NombreI
  FicheroI = NombreI 'nuevo fichero índice creado

  'Ocultar la form
  NombreFg.Text = ""
  NombreFg.SetFocus
  GuardarF.Hide
```



```
  Form1.Clave.SetFocus
Salir:
  Exit Sub
RutinaDeError:
  If Err = 53 Then Resume Next 'fichero no encontrado
  MsgBox "Error en el acceso al fichero", 48, "Libros"
  NombreFg.SetFocus
  Resume Salir
End Sub
```

Este procedimiento primero compone el nombre del fichero índice con el mismo nombre base que el fichero de datos y extensión *.ind*. Después comprueba si el fichero especificado existe, en cuyo caso se lo notifica al usuario. A continuación, se utiliza la sentencia **Name** para asignar los nuevos nombres a los ficheros. Recuerde que cuando se trata de un fichero nuevo, los nombres asignados inicialmente son *Temp.dat* y *Temp.ind*. Por último, se enfoca la caja de texto *NombreFg* para una siguiente entrada y se oculta la forma.

Ahora abra la ventana de código correspondiente a la orden *Cancelar* y escriba el código que se muestra a continuación.

```
Sub Cancelar_Click ()
  NombreFg.Text = ""
  NombreFg.SetFocus
  GuardarF.Hide
  Form1.Clave.SetFocus
End Sub
```

Cuando el usuario quiera visualizar los datos correspondientes a un determinado libro, tiene que escribir la clave en la caja de texto *Clave* y a continuación ejecutar la orden *Buscar Registro* del menú *Fichero*. Para realizar este proceso, abra la ventana de código correspondiente a esta orden y escriba el código que se muestra a continuación.

```
Sub BuscarReg_Click ()
  Dim vr As Integer
  vr = VisualizarRegistro()
End Sub
```

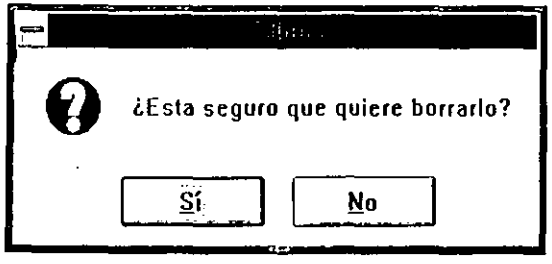
Para visualizar en la forma *Form1* los datos correspondientes a un determinado registro del fichero de datos, este procedimiento llama a la función *VisualizarRegistro* que se muestra a continuación. Para crear esta función ejecute la orden **New Procedure** del menú **Code** y escriba el código mostrado a continuación.

```
Function VisualizarRegistro () As Integer
  Dim I As Integer
```

```
    'Buscar clave en el fichero índice
    I = HashOut(Val(Form1.Clave.Text))
    If I = 0 Then 'no existe la clave
      Exit Function
      VisualizarRegistro = 0
    End If
    'Visualizar registro
    Open FicheroD For Random As #1 Len = Len(Libros)
    Get #1, I, Libros
    Close #1
    Form1.Clave.Text = Str$(Libros.Clave)
    Form1.Título.Text = Libros.Título
    Form1.Autor.Text = Libros.Autor
    Form1.Editorial.Text = Libros.Editorial
    Form1.Prestado.Text = Libros.Prestado
    If Libros.Borrado = -1 Then
      MsgBox "Este registro está borrado", 48, "Libros"
      VisualizarRegistro = 0
    Else
      VisualizarRegistro = I
    End If
End Function
```

Esta función invoca a la función *HashOut* para obtener la posición del registro en el fichero de datos que tiene la clave especificada. A continuación, lee el registro y lo visualiza. También, si el registro está marcado como borrado envía un mensaje notificándolo. La función *VisualizarRegistro* devuelve el número del registro visualizado, o un 0 si la clave especificada no se encuentra.

Cuando el usuario quiera borrar un determinado registro, tiene que escribir la clave en la caja de texto *Clave* y a continuación ejecutar la orden *Borrar Registro* del menú *Fichero*. Entonces, si la clave existe, se presenta el registro sobre la forma *Form1* y se visualiza la siguiente caja de diálogo.



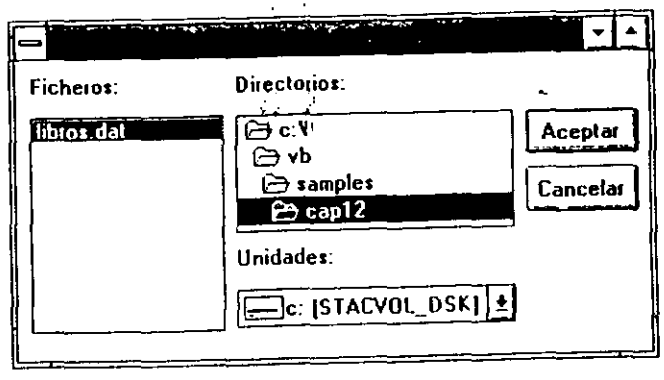
Si el usuario pulsa el botón *Sí*, la función **MsgBox** que da lugar a esta caja devuelve el valor 6 (y el valor 7 si se pulsa *No*) en cuyo caso, el registro especificado se marca como borrado. Para ello, se lee el registro, se asigna el valor −1 a su campo *Borrado*, y se vuelve a escribir en la misma posición.

Abra la ventana de código correspondiente a la orden *Borrar Registro* del menú *Fichero* y escriba el código mostrado en el siguiente procedimiento.

```
Sub BorrarReg_Click ()
  Dim Cadena As String, I As Integer
  I = VisualizarRegistro()
  If I <> 0 Then
    Cadena = "¿Esta seguro que quiere borrarlo?"
    If MsgBox(Cadena, 36, "Libros") = 6 Then
      Open FicheroD For Random As #1 Len = Len(Libros)
      Get #1, I, Libros
      Libros.Borrado = -1
      Put #1, I, Libros
      Close #1
    End If
  End If
End Sub
```

Cuando se ejecuta la orden *Abrir* aparecerá la caja de diálogo que se muestra en la siguiente figura. Para ello, abra la ventana de código correspondiente a la orden *Abrir* y escriba el código que se muestra a continuación.

```
Sub FicheroAbrir_Click ()
  SistemaF.Show
End Sub
```



Como el diseño de esta misma caja ya fue explicado en el Capítulo 8 "Ficheros de datos", no vamos a volverlo a hacer aquí. Simplemente, por comodidad para usted mostraremos los procedimientos implicados en el diseño de dicha caja. También, si usted ya dispone de este fichero *.frm* (en nuestra aplicación

se denomina *sistdefl.frm*) puede añadirlo a esta aplicación y realizar las modificaciones que necesite en función del código que a continuación se detalla.

Si desea visualizar sólo los ficheros de la lista de ficheros *File1* con una determinada extensión, por ejemplo **.dat*, ponga la propiedad **Pattern** a ese valor.

Añada las siguientes declaraciones a la sección de declaraciones de la forma *Sistema de Ficheros*.

```
Const CLIC_EN_DIR = 1, CLIC_EN_FILE = 2
Dim ÚltimoCambio As Integer
Dim Texto As String
```

Añada el siguiente código al procedimiento *Drive1_Change* de la forma *Sistema de Ficheros*.

```
Sub Drive1_Change ()
  On Error GoTo Driver
  Dir1.Path = Drive1.Drive
  Exit Sub
Driver:
  MsgBox "Error: unidad no preparada", 48, "Editor"
  Exit Sub
End Sub
```

Añada el siguiente código al procedimiento *Dir1_Click* de la forma *Sistema de Ficheros*.

```
Sub Dir1_Click ()
  ÚltimoCambio = CLIC_EN_DIR
End Sub
```

Añada el siguiente código al procedimiento *File1_Click* de la forma *Sistema de Ficheros*.

```
Sub File1_Click ()
  ÚltimoCambio = CLIC_EN_FILE
End Sub
```

Añada el siguiente código al procedimiento *Aceptar_Click* de la forma *Sistema de Ficheros*.

```
Sub Aceptar_Click ()
  Dim NumRegsFich As Integer, I As Integer
  Aceptar.SetFocus
```



```
  Select Case ÚltimoCambio
    Case CLIC_EN_DIR
      Dir1.Path = Dir1.List(Dir1.ListIndex)
    Case CLIC_EN_FILE
      If (Right$(Dir1.Path, 1) = "\") Then
        FicheroD = Dir1.Path + File1.FileName
      Else
        FicheroD = Dir1.Path + "\" + File1.FileName
      End If
      FicheroI = Left$(FicheroD, Len(FicheroD) - 3) + "ind"
      SistemaF.Hide
  End Select
End Sub
```

Este procedimiento compone el nombre del fichero índice a partir del nombre del fichero de datos sin extensión. El fichero índice tiene el mismo nombre base que el fichero de datos y extensión *.ind*.

Añada el siguiente código al procedimiento *Cancelar_Click* de la forma *Sistema de Ficheros*.

```
Sub Cancelar_Click ()
  SistemaF.Hide
End Sub
```

Cuando el usuario quiera finalizar la aplicación ejecutará la orden *Salir* del menú *Fichero*. Abra la ventana de código correspondiente a esta orden y escriba el código que se muestra a continuación.

```
Sub Salir_Click ()
  End
End Sub
```

Guarde la aplicación y ejecútela. La razón de esta aplicación es simplemente que usted tenga una base de conocimientos para trabajar con ficheros indexados.

IMPRESIÓN DE RESULTADOS

En muchas ocasiones necesitará obtener resultados impresos. La forma más cómoda de realizar esta operación es utilizando el objeto **Printer**. Su sintaxis es,

Printer.*propiedad* [= *expresión*]
Printer.*método* [*expresión*]

El objeto **Printer** es utilizado para controlar el texto y los gráficos que se imprimen en una página y para enviarlos directamente a la impresora por defecto. Con este objetivo, **Printer** tiene asociados varias propiedades y varios métodos que usted podrá ver con detalle en la ayuda suministrada por Visual Basic. Las tablas siguientes muestran algunas de las propiedades y métodos más usuales.

Propiedad	Descripción
CurrentX	Pone o retorna la coordenada X.
CurrentY	Pone o retorna la coordenada Y.
FontSize	Pone o retorna el tamaño de la fuente utilizada para imprimir el texto.
Fontestilo	Determina el estilo de la fuente (**Bold**, **Italic**, **Strikethru**, **Transparent**, **Underline**)
ScaleMode	Pone o retorna las unidades para un sistema de coordenadas. Por ejemplo: 1 – Twips (valor por defecto), 2 – Puntos, 3 – Pixels, 4 – Caracteres.
ScaleWidth	Pone o retorna el ancho del eje horizontal del sistema de coordenadas de un determinado objeto.

Método	Descripción
Print	Escribe un expresión numérica o alfanumérica.
NewPage	Finaliza la página actual y avanza a una nueva página.
EndDoc	Finaliza un documento enviado a la impresora, descargándolo en dicho dispositivo o en la cola de impresión. Si se invoca **EndDoc** inmediatamente después del método **NewPage**, entonces no se añade una página en blanco.

Como ejemplo, añada al menú *Fichero*, la orden titulada *Imprimir Fichero* y de nombre *Imprimir*. Cuando el usuario ejecute esta orden se realizará un listado por la impresora de todos los registros que no estén borrados, imprimiendo por cada uno de ellos los campos *Clave*, *Título* y *Editorial*. Para ello, abra la ventana de código correspondiente a esta orden y escriba el procedimiento que se muestra a continuación.

```
Sub Imprimir_Click ()
  'Imprimir los registros del fichero de datos
  'que no estén marcados como borrados
  Dim Cabl As String
  Dim NumRegs As Integer, I As Integer
  Cabl = "Listado de libros"
  Open FicheroD For Random As #1 Len = Len(Libros)
  P    'er.ScaleMode = 4 'caracteres
```



```
  Printer.NewPage 'nueva página
  Printer.FontSize = 24 'tamaño del carácter en puntos
  Printer.CurrentX = Printer.ScaleWidth \ 2 - Len(Cabl) \ 2
  Printer.CurrentY = 6
  Printer.FontBold = -1 'negrita
  Printer.FontUnderline = -1
  Printer.Print Cabl 'cabecera centrada, negrita y subrayada
  Printer.FontUnderline = 0
  Printer.FontBold = 0
  Printer.FontSize = 12
  Printer.Print
  NumRegs = LOF(1) \ Len(Libros) 'total registros
  For I = 1 To NumRegs
    Get #1, , Libros
    If Not Libros.Borrado Then
      Printer.Print Libros.Clave,
                    Libros.Título,
                    Tab(50); Libros.Editorial
    End If
  Next
  Close #1
  Printer.EndDoc 'finaliza documento
End Sub
```

CAPÍTULO 13

INTERCAMBIO DINÁMICO DE DATOS

INTRODUCCIÓN

El intercambio dinámico de datos (**Dynamic data exchange**, abreviadamente **DDE**) es un mecanismo soportado por Visual Basic que permite que una aplicación hable continuamente con otra para intercambiarse datos automáticamente. DDE automatiza el intercambio manual entre aplicaciones que se realiza a través del portapapeles. No todas las aplicaciones soportan DDE.

Por ejemplo, utilizando DDE el procesador de textos **Word** para **Windows** puede hablar con la hoja de cálculo **Excel** para **Windows**. Esto permitirá por ejemplo, que una hoja de cálculo importada desde **Excel** a nuestro documento en **Word**, se modifique automáticamente cuando en la hoja de cálculo realicemos cualquier variación.

COMO TRABAJA EL DDE

Un DDE es similar a una conversación entre dos personas. La aplicación que inicia la conversación se denomina *cliente*; la aplicación que responde a su cliente se denomina *servidor*. Una aplicación puede estar ocupada en varias conversaciones simultáneamente, actuando en unas como cliente y en otras como servidor.

En Visual Basic pueden actuar como *cliente*, las cajas de texto, las cajas de imagen y las etiquetas, mientras que como *servidor* sólo pueden actuar las formas.

Cuando un *cliente* inicia una conversación DDE, debe especificar dos cosas:

- El nombre del *servidor* con el que quiere hablar (aplicación que responde).
- El tema de la conversación (datos a intercambiar).

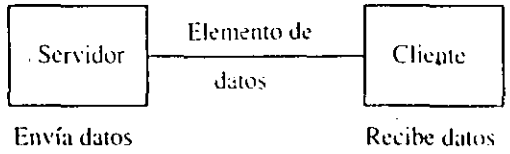
Cuando un *servidor* recibe una petición de conversación referente a un tema, la reconoce y en caso afirmativo inicia la conversación. En general, es el *servidor* el que envía datos al *cliente*.

Cada aplicación que puede actuar como *servidor* tiene un único nombre que normalmente coincide con el nombre del fichero ejecutable sin extensión. Por ejemplo, **Excel**, **WinWord**. Si el *servidor* es una aplicación Visual Basic, su nombre es el dado al fichero ejecutable correspondiente a la misma. Para otras aplicaciones Windows vea la documentación correspondiente a la misma.

El tema de conversación se corresponde con la unidad de datos reconocida por el *servidor*. Por ejemplo, **Excel** reconoce ficheros con extensión .XLS o .XLC, mientras que **WinWord** reconoce ficheros con extensión .DOC. Cuando el *servidor* de una conversación es una forma correspondiente a una aplicación Visual Basic, el tema queda especificado por la propiedad **LinkTopic**.

Durante una conversación se intercambian elementos de datos pertenecientes a un tema. Por ejemplo, **Excel** reconoce celdas referenciadas por su línea (L) y columna (C) tal como L1C1. Cuando el *cliente* de una conversación es un control de una aplicación Visual Basic, el elemento de datos que se pasa durante la conversación queda especificado por la propiedad **LinkItem**. Cuando el *servidor* de una conversación es una forma correspondiente a una aplicación Visual Basic, el elemento de datos que se pasa durante la conversación es el nombre de una caja de texto, de una etiqueta, o de una caja de imagen, perteneciente a la forma.

Una conversación DDE constituye un enlace entre las dos aplicaciones que mantienen la conversación. Hay dos formas de establecer un enlace entre aplicaciones:



- *Enlace frío*. El *servidor* suministra nuevos datos solamente cuando el *cliente* lo requiere. Para realizar esta petición y así actualizar los datos, será necesario utilizar el método **LinkRequest**. Este tipo de enlace no puede establecerse en tiempo de diseño.
- *Enlace caliente*. El *servidor* suministra nuevos datos al cliente cada vez que el elemento de datos definido por la propiedad **LinkItem** cambia.

En una aplicación Visual Basic, el tipo de enlace para un objeto queda especificado por la propiedad **LinkMode**. Para un control utilizado como *cliente*, el valor de esta propiedad puede ser: 0, no hay enlace (es el valor por defecto); 1, enlace caliente; 2, enlace frío. Para una forma utilizada como *servidor* el valor de esta propiedad puede ser: 0, no hay enlace; 1, servidor (es el valor por defecto).

CREAR UN ENLACE EN TIEMPO DE DISEÑO

Durante el diseño de una aplicación es posible establecer enlaces entre ésta aplicación y otras aplicaciones que soporten DDE. Estos enlaces son guardados en las propiedades **Link** de las formas y de los controles de la aplicación. Dichos enlaces serán establecidos automáticamente siempre que la aplicación se ejecute.

Aplicación Visual Basic como cliente

Cuando quiera que su aplicación Visual Basic reciba datos de otra aplicación, tiene que establecer un enlace definiendo su aplicación como cliente y la otra como servidor.

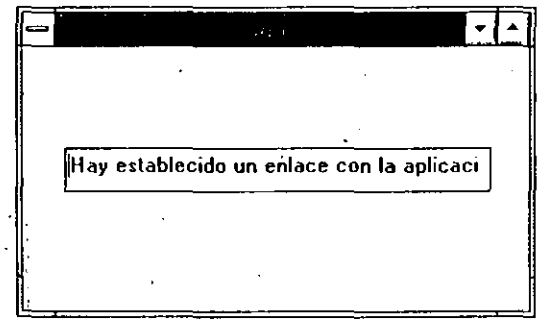
Como ejemplo, vamos a establecer un enlace caliente entre la aplicación **WinWord** (servidor) y una aplicación Visual Basic que denominaremos *DdeWinW* (cliente).

Inicie una nueva aplicación y cree una forma titulada *Cliente* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *DdeWinW.mak* y la forma con el nombre *DdeWinW.frm*.

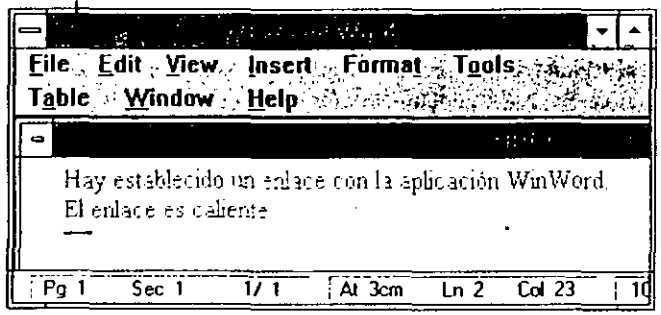
Los valores de las propiedades **Link** de la caja de texto de la tabla siguiente, son los valores por defecto. Los valores necesarios para realizar el enlace caliente con el servidor, serán puestos automáticamente por Visual Basic cuando se cree dicho enlace.

Objeto	Propiedad	Valor
Forma	FormName Caption	Form1 Cliente
Caja de texto	CtlName LinkItem LinkMode LinkTimeout LinkTopic Text	Cliente (nada) 0 (ninguno) 50 (nada) (nada)

La propiedad **LinkTimeout** determina la cantidad de tiempo que un control (cliente) espera para establecer la conversación con el servidor. Se mide en décimas de segundo y el valor por defecto es 50. Si el tiempo de respuesta del servidor supera este valor, se genera un mensaje de error. Un valor de −1 significa esperar indefinidamente. En este caso, podemos finalizar la espera pulsando la tecla **Alt**.



A continuación ejecute la aplicación Word para Windows (**WinWord**), cree un documento y asígnele el nombre *midocu.doc*.



Para crear un enlace caliente entre una aplicación Visual Basic como cliente y otra aplicación como servidor, en este caso entre *DdeWinW* y **WinWord**, ejecute los siguientes pasos:

1. Seleccione en la otra aplicación, los datos con los que quiere establecer el enlace. En el ejemplo, seleccione todo el documento *midocu.doc* o una parte del mismo.
2. Ejecute la orden **Copy** (Copiar) del menú **Edit** (Edición) de la otra aplicación, para copiar en el portapapeles los datos seleccionados.
3. Ahora diríjase a la aplicación Visual Basic (cliente) y seleccione el control que va a recibir los datos. En el ejemplo, la caja de texto *Cliente*.
4. Ejecute la orden **Paste Link** del menú **Edit** de Visual Basic.

Observará que en la caja de texto, aparece el texto seleccionado. Esto quiere decir que el enlace está establecido. Vaya al procesador de textos y modifique el texto. Vuelva a la aplicación de Visual Basic y observe como los cambios se han efectuado. Recuerde que esto es un enlace caliente.

El enlace establecido es guardado con la forma y es permanente. Por lo tanto, siempre que abra la aplicación, tanto en tiempo de diseño como en tiempo de ejecución, ésta intentará restablecer la conversación con el servidor. Si no hay una aplicación que reconozca el tema de conversación, Visual Basic visualizará un mensaje indicándolo.

Ahora verifique las propiedades de la caja de texto descritas anteriormente. Observará que sus valores han sido cambiados por el hecho de crear el enlace. Estos valores se presentan en la tabla siguiente.

Caja de texto	CtlName LinkItem LinkMode LinkTimeout LinkTopic Text	Cliente DDE_LINK1 1 - Hot 50 WinWord|...\MIDOCU.DOC (nada)

Analizando los valores de las propiedades deducimos que la caja de texto es un cliente que tiene establecido un enlace caliente (**LinkMode**) para conversar sobre el tema MIDOCU.DOC con el servidor **WinWord** (**LinkTopic**). El elemento de datos utilizado por el servidor para enviar los datos es DDE_LINK1. Este nombre referencia el texto seleccionado.

Aplicación Visual Basic como servidor

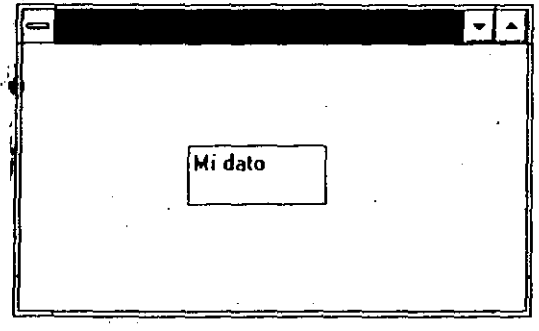
Cuando quiera que su aplicación Visual Basic envíe datos a otra aplicación, tiene que establecer un enlace definiendo su aplicación como servidor y la otra como cliente.

Como ejemplo, vamos a establecer un enlace caliente entre la hoja de cálculo **Excel** (cliente) y una aplicación Visual Basic que denominaremos *DdeExcel* (servidor).

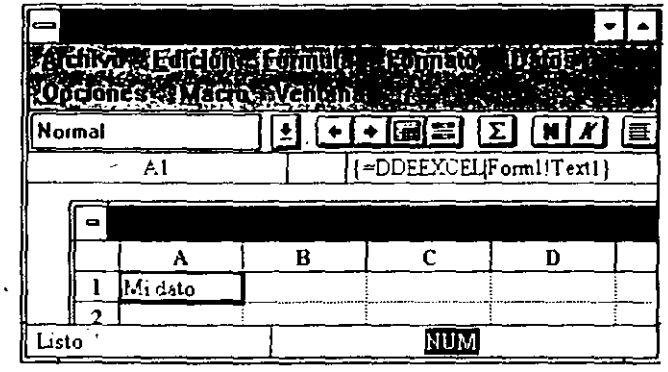
Inicie una nueva aplicación y cree una forma titulada *Servidor* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *DdeExcel.mak* y la forma con el nombre *DdeExcel.frm*.

Los valores de las propiedades **Link** de la forma de la tabla siguiente, son los valores por defecto.

Objeto	Propiedad	Valor
Forma	FormName	Form1
	Caption	Servidor
	LinkMode	1 - Server
	LinkTopic	Form1
Caja de texto	CtlName	Text1
	Text	Mi dato



A continuación ejecute la aplicación **Excel** para Windows.



Para crear un enlace entre su aplicación Visual Basic como servidor y otra aplicación como cliente, en este caso entre *DdeExcel* y **Excel**, ejecute los siguientes pasos:

1. Diríjase a su aplicación Visual Basic (servidor) y seleccione el control que contiene los datos que quiere enviar. En el ejemplo, seleccione el control *Text1*.
2. Ejecute la orden **Copy** del menú **Edit** de Visual Basic, para copiar en el portapapeles los datos seleccionados.
3. Cambie a la otra aplicación (cliente) y seleccione el destino de los datos. En el ejemplo, seleccione la celda *A1* de la hoja de cálculo **Excel**.
4. Ejecute la orden **Paste Link** (Pegar vínculos) del menú **Edit** (Edición) de la otra aplicación. En el ejemplo, de la hoja de cálculo **Excel**.

Observará que en la celda *A1* elegida aparece el contenido de la caja de texto. Esto quiere decir que el enlace está establecido. Vaya a la aplicación Visual Basic, elija la propiedad **Text** de la caja de texto y modifique su contenido. Observe como los cambios se hacen también en la hoja de cálculo.

El enlace establecido es guardado con la forma y es permanente. Sin embargo, cuando ejecute la aplicación, Visual Basic tiene que romper el enlace para pasar de tiempo de diseño a tiempo de ejecución. Algunas aplicaciones son capaces de restablecer automáticamente sus conversaciones DDE, pero hay otras que no. Estas aplicaciones que soportan DDE pero que no son capaces de restablecer sus conversaciones automáticamente, proveen alguna forma manual de restablecer dichas conversaciones. Para tener acceso a esta información, consulte la documentación de esa aplicación.

Por ejemplo, ejecute la aplicación *DdeExcel* que acabamos de desarrollar y modifique el contenido de la caja de texto. Observe que dichas modificaciones no se reflejan en la hoja de cálculo. Quiere esto decir que el enlace no se ha restablecido. Diríjase a la hoja de cálculo y proceda a restablecerlo manualmente ejecutando la orden **Paste Link** (Pegar vínculos) del menú **Edit** (Edición).

Si en lugar de realizar el ejemplo con **Excel** lo realiza con **Word** le ocurrirá lo mismo. En este caso para restablecer el enlace con el servidor tendrá que ejecutar la orden **Links** (Vínculos) del menú **Edit** (Edición).

Ahora verifique las propiedades de la caja de texto descritas anteriormente. Observará que sus valores no se han modificado. Estos valores se presentan en la tabla siguiente.

Forma	FormName	Form1
	Caption	Servidor
	LinkMode	1 - Server
	LinkTopic	Form1

Analizando los valores de las propiedades y la expresión asignada a la celda *A1, {=DDEEXCEL\|Form1!Text1}*, deducimos que la hoja de cálculo es un cliente que tiene establecido un enlace caliente para conversar sobre el tema *Form1!Text1* con el servidor *DDEEXCEL*, que es la aplicación Visual Basic que contiene la forma *Form1*. El elemento de datos es *Text1*.

CREAR UN ENLACE EN TIEMPO DE EJECUCIÓN

Hemos visto que muchas aplicaciones que soportan DDE proveen una forma estándar para que el usuario pueda establecer una conversación DDE sin escribir código. Por ejemplo, Visual Basic permite hacer esto con las órdenes **Copy** y **Paste Link** del menú **Edit**. También, como vamos a ver a continuación, es posible establecer un enlace entre dos aplicaciones, escribiendo código. En este caso el enlace puede ser caliente o frío.

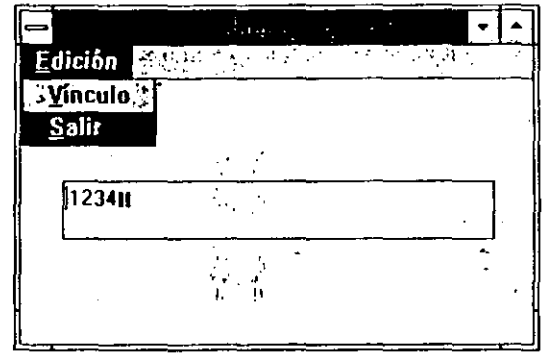
Aplicación Visual Basic como cliente

Cuando quiera que su aplicación Visual Basic reciba datos de otra aplicación, tiene que establecer un enlace definiendo su aplicación como cliente y la otra como servidor.

Como ejemplo, vamos a establecer un enlace caliente entre la aplicación **Excel** (servidor) y una aplicación Visual Basic que denominaremos *Dde01* (cliente).

Inicie una nueva aplicación y cree una forma titulada *Cliente* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *Dd01.mak* y la forma con el nombre *Dde01.frm*.

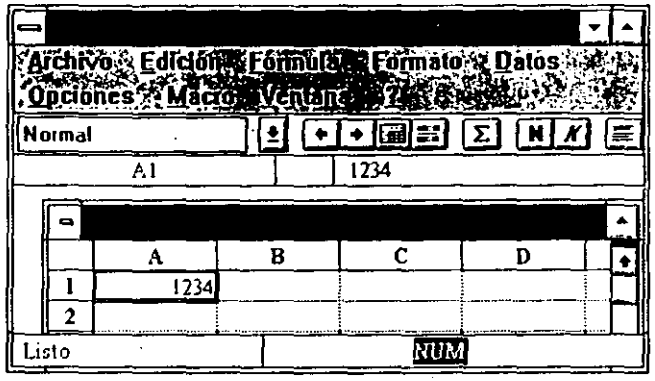
Objeto	Propiedad	Valor
Menú	Caption	&Edición
	CtlName	Edit
Menú: orden 1	Caption	&Vínculo
	CtlName	EditVinculo
Menú: orden 2	Caption	&Salir
	CtlName	EditSalir
Forma	FormName	Form1
	Caption	Cliente
Caja de texto	CtlName	Cliente
	Text	(nada)



En primer lugar cargue el fichero CONSTANT.TXT en el módulo global, ya que en él están definidas constantes como NONE, HOT, etc., que vamos a utilizar en nuestra aplicación. Abra la ventana de código para el módulo *Global.bas* y ejecute la orden **Load Text** del menú **Code**; escriba el nombre ...\vb\constant.txt y pulse el botón *Replace*.

Cuando el usuario haga clic en la orden *Vínculo* del menú *Edición* de nuestra aplicación, se generará un suceso **Click**. La respuesta a este suceso va a ser esta-

blecer un enlace caliente entre el servidor **Excel** y el cliente *Text1*. Los datos serán enviados desde la celda L1C1 (A1) de **Excel** a la caja de texto *Text1*.



Para crear el enlace hay que poner las propiedades **Link** del cliente *Text1* a los valores que se indican a continuación:

- Inicialmente, la propiedad **LinkMode** de *Text1* se pone a cero (NONE) con el fin de romper cualquier enlace existente.
- A continuación, se asigna un valor de la forma "*aplicación_servidor|tema*" a la propiedad **LinkTopic** de *Text1* (el carácter | es el ASCII 124): el nombre del fichero ejecutable de la *aplicación* que trabaja como servidor es **Excel** y el *tema* es un fichero correspondiente a la hoja de cálculo desde la cual se van a enviar los datos; supongamos que se denomina *Hoja1*.
- Después, mediante la propiedad **LinkItem** se especifica el elemento de datos, L1C1 (A1), utilizado para la transferencia.
- Y por último, conocidos el servidor, el tema de conversación y el elemento de datos, se establece un enlace caliente.

Estas operaciones son las que tiene que realizar el procedimiento *EditVínculo_Click* que se presenta a continuación.

```
Sub EditVínculo_Click ()
  Text1.LinkMode = NONE 'romper enlace
  Text1.LinkTopic = "Excel|Hoja1"
  Text1.LinkItem = "L1C1"
  Text1.LinkMode = HOT 'enlace caliente
End Sub
```


La orden *Salir* finaliza la aplicación. El procedimiento correspondiente se presenta a continuación:

```
Sub EditSalir_Click ()
  End
End Sub
```

Guarde la aplicación *Dde01.mak* y asegúrese que tiene cargado el fichero *Hoja1* en la aplicación **Excel**. Ponga en marcha su aplicación y ejecute la orden *Vínculo* del menú *Edición* de la misma. Cuando se establezca el enlace entre ambas aplicaciones, observará que el dato de la celda *A1* aparece en la caja de texto *Text1*. Ahora, cualquier variación que realice en la celda *A1* se reflejará en la caja de texto *Text1*.

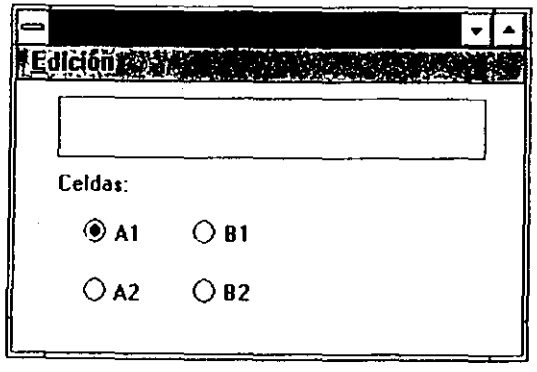
Una vez establecido el enlace, puede cambiar si quiere el elemento de datos especificado por la propiedad **LinkItem**, sin que esta operación afecte a dicho enlace.

Como ejemplo, supongamos que deseamos tener acceso a los datos de las celdas *A1*, *A2*, *B1* y *B2*.

Modifique la forma anterior, añadiendo los controles que se indican en la tabla siguiente.

Objeto	Propiedad	Valor
Opción 1	Caption	A1
	CtlName	LnCn
	Index	0
	Visible	True
Opción 2	Caption	B1
	CtlName	LnCn
	Index	1
	Visible	False
Opción 3	Caption	A2
	CtlName	LnCn
	Index	2
	Visible	False
Opción 4	Caption	B2
	CtlName	LnCn
	Index	3
	Visible	False

Observe que el conjunto de opciones forma el array de controles *LnCn*.



Cuando el usuario haga clic en una de estas opciones, se genera el suceso **Click** para el control *LnCn(Index)*. Por lo tanto, la respuesta a este suceso la daremos en el procedimiento *LnCn_Click* que tiene que visualizar el valor correspondiente a la celda referenciada. Para ello, lo único que tenemos que hacer es modificar el valor de la propiedad **LinkItem** en función de la celda elegida (opción elegida).

```
Sub LnCn_Click (Index As Integer)
  Select Case Index
    Case 0
      Text1.LinkItem = "L1C1"   'celda A1
    Case 1
      Text1.LinkItem = "L1C2"   'celda B1
    Case 2
      Text1.LinkItem = "L2C1"   'celda A2
    Case 3
      Text1.LinkItem = "L2C2"   'celda B2
  End Select
End Sub
```

Guarde la aplicación, ejecútela y compruebe los resultados.

Aunque hasta ahora hemos elegido una celda como elemento de datos, esto no quiere decir que siempre sea así. Por ejemplo, el elemento de datos puede ser el rango de celdas L1C1:L3C3. Si el elemento de datos se cambia durante la ejecución, diremos exactamente lo mismo.

```
Text1.LinkItem = "L1C1:L3C3"
```


Establecer un enlace frío

Mientras que en tiempo de diseño el único tipo de enlace que se puede realizar es un enlace caliente, en tiempo de ejecución se puede realizar además del anterior, otro tipo de enlace denominado enlace frío. Entre ambos existe una diferencia importante: con un enlace caliente, una variación en los datos del servidor es reflejada automáticamente en el cliente. Si embargo, con un enlace frío cuando los datos varían en el servidor, la actualización de los datos en el cliente no sucede a menos que se solicite explícitamente con el método **LinkRequest**.

En muchos casos, un enlace frío puede ser mejor que un enlace caliente. Por ejemplo, volviendo a la hoja de cálculo, suponga que debido a los cálculos las celdas enlazadas están sometidas a numerosas variaciones hasta completar dichos cálculos. Con un enlace caliente, estas variaciones se están reflejando también en el cliente con la consiguiente pérdida de velocidad en la aplicación, cosa que no ocurre si realiza un enlace frío.

En una conversación DDE, el método **LinkRequest** permite actualizar el contenido de un control que actúa como cliente con los datos enviados desde el servidor. Su sintaxis es,

control.**LinkRequest**

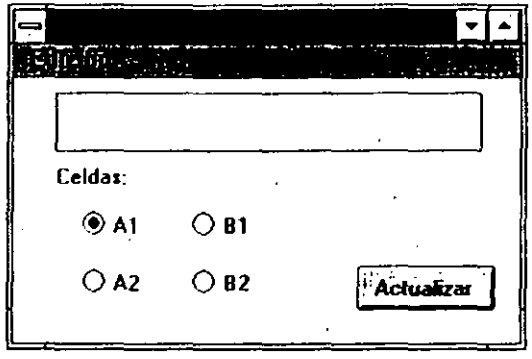
donde *control* es una caja de texto, una caja de imagen o una etiqueta que actúa como cliente en la conversación DDE.

Como ejemplo, vamos a establecer un enlace frío entre la aplicación **Excel** (servidor) y la aplicación Visual Basic anterior denominada *Dde01* (cliente).

A la hora de crear el enlace la única variación que hay que realizar es cambiar el valor de la propiedad **LinkMode** para que especifique un enlace frío.

```
Sub EditVínculo_Click ()
  Text1.LinkMode = NONE 'romper enlace
  Text1.LinkTopic = "Excel|Hoja1"
  Text1.LinkItem = "L1C1"
  Text1.LinkMode = COLD 'enlace frío
End Sub
```

A continuación añada a la forma un botón *Actualizar* de manera que cuando el usuario haga clic sobre él, los datos de la caja de texto queden actualizados.



El procedimiento asociado con el botón *Actualizar* sólo tiene que ejecutar el método **LinkRequest** para el control *Text1*, si hay un enlace frío establecido.

```
Sub Actualizar_Click ()
  If Text1.LinkMode = COLD Then
    Text1.LinkRequest
  End If
End Sub
```

Guarde la aplicación, ejecútela y compruebe los resultados.

Aplicación Visual Basic como servidor

Si quiere que su aplicación Visual Basic envíe datos a otra, tiene que establecer un enlace definiendo su aplicación como servidor y la otra como cliente.

Aunque el flujo de datos en una conversación DDE es generalmente desde la aplicación servidor a la aplicación cliente, el cliente puede también enviar datos a la aplicación servidor, utilizando el método **LinkPoke**.

En una conversación DDE, el método **LinkPoke** permite transferir el contenido de un control a la aplicación servidor. Su sintaxis es,

control.**LinkPoke**

donde *control* es una caja de texto, una caja de imagen o una etiqueta que actúa como cliente en la conversación DDE.

Como ejemplo, vamos a establecer un enlace frío o caliente entre una aplicación Visual Basic que denominaremos *Servidor* y otra aplicación Visual Basic que denominaremos *Cliente*.

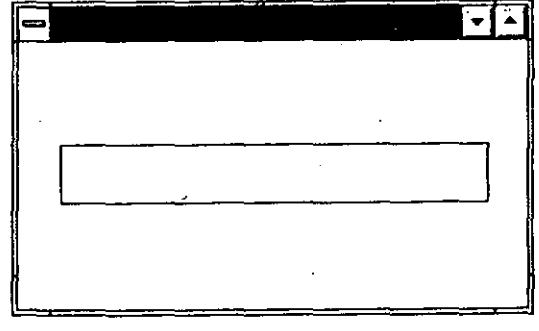
Para resolver este problema tiene que:

- Crear una aplicación Visual Basic que funcione como servidor.
- Crear una aplicación Visual Basic que funcione como cliente.
- Establecer un enlace frío para DDE entre la aplicación cliente y la aplicación servidor.
- Utilizar el método **LinkRequest** para actualizar la información de la aplicación cliente enviada desde la aplicación servidor.
- Establecer un enlace caliente para DDE entre la aplicación cliente y la aplicación servidor.
- Utilizar el método **LinkPoke** para enviar información desde la aplicación cliente a la aplicación servidor.

Inicialmente la aplicación cliente envía órdenes a la aplicación servidor para establecer el enlace. Después a través de una conversación DDE, el servidor envía datos al cliente o acepta datos del cliente a petición de éste.

A continuación desarrollamos paso a paso la aplicación enunciada. En primer lugar creamos una aplicación Visual Basic que funcione como servidor.

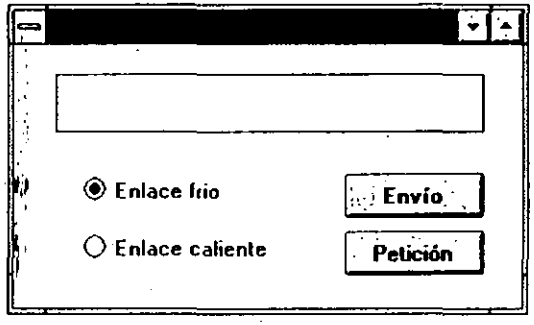
Inicie una nueva aplicación y cree una forma titulada *Servidor*. Observe que los valores por defecto de las propiedades **LinkMode** y **LinkTopic** son *1 (Server)* y *Form1* respectivamente. Esto quiere decir que las aplicaciones que utilicen ésta como servidor, tienen que buscar para realizar el enlace una aplicación con un tema de conversación *"Servidor|Form1"*. Añada a la forma una caja de texto de nombre *Datos*. Guarde la aplicación con el nombre *Servidor.mak* y la forma con el nombre *Servidor.frm*.



Ahora cree un fichero ejecutable SERVIDOR.EXE para esta aplicación. Para ello, ejecute la orden **Make EXE File** del menú **File**.

En segundo lugar cree una aplicación Visual Basic que funcione como cliente. Inicie una nueva aplicación y cree una forma titulada *Cliente* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *Cliente.mak* y la forma con el nombre *Cliente.frm*.

Objeto	Propiedad	Valor
Caja de texto	CtlName Text	Text1 (nada)
Opción 1	Caption CtlName Visible	Enlace frío EnlaceFrio True
Opción 2	Caption CtlName Visible	Enlace caliente EnlaceCaliente False
Botón 1	Caption CtlName	Envío Envío
Botón 2	Caption CtlName	Petición Petición



Cargue el fichero CONSTANT.TXT en el módulo global, ya que en él están definidas constantes como NONE, HOT, etc., que vamos a utilizar en nuestra aplicación. Abra la ventana de código para el módulo *Global.bas* y ejecute la orden **Load Text** del menú **Code**; escriba el nombre ...\vb\constant.txt y pulse el botón *Replace*. Cámbielo de nombre y denomínelo *Cliente.bas*.

Cuando se inicie la aplicación *Cliente* debe iniciarse también la aplicación *Servidor* que hemos creado anteriormente, puesto que el siguiente paso es crear un enlace entre ambas aplicaciones. Para ello, escriba el siguiente procedimiento conducido por el suceso **Load** que se encargará de arrancar la aplicación *Servidor* y de crear el enlace entre ambas:

```
Sub Form_Load ()
  'Este procedimiento arranca la aplicación VB
  'que funciona como servidor
  Dim Id As Integer

  Id = Shell("c:\vb\samples cap13\servidor", 1)
  'La siguiente sentencia asegura que el Shell
  'finalizará antes de intentar establecer el enlace
  Id = DoEvents()

  Text1.LinkMode = NONE 'romper enlace si existe
  Text1.LinkTopic = "Servidor|Form1"
  Text1.LinkItem = "Datos"
  Text1.LinkMode = COLD 'enlace frío por defecto
End Sub
```

La función **DoEvents** interrumpe Visual Basic y cede el control al entorno operativo (por ejemplo, Windows), hasta que todos los mensajes que están esperando en la cola de procesos sean ejecutados.

En el procedimiento anterior, la función **DoEvents** garantiza que la ejecución de la aplicación *Servidor* finalice antes de continuar con el proceso actual. Si no se llama a la función **DoEvents**, las sentencias que crean el enlace frío entre el cliente y el servidor se ejecutarán antes de que el proceso de carga de éste haya finalizado, lo que provocará un error, ya que la petición de conversación del cliente al servidor no puede ser contestada por no estar éste activo en el tiempo establecido por defecto. En este caso concreto, otra solución sería incrementar el tiempo por defecto especificado por la propiedad **LinkTimeout**.

Como el servidor de la conversación es una forma correspondiente a una aplicación Visual Basic, el elemento de datos utilizado por el servidor para enviar los datos es su caja de texto *Datos*. Recuerde que éste debe ser una caja de texto, una etiqueta, o una caja de imagen, perteneciente a la forma.

Una vez enlazadas las dos aplicaciones, si quiere puede modificar el tipo de enlace. Esta es la finalidad de las dos opciones que aparecen en la forma *Cliente*.

La opción *Enlace frío* establece un enlace frío entre el cliente y el servidor. El procedimiento conducido por el suceso **Click** para este control es el siguiente:

```
Sub EnlaceFrio_Click ()
  Petición.Visible = TRUE 'hacer visible este botón
  Text1.LinkMode = NONE 'romper en enlace
  Text1.LinkMode = COLD 'enlace frío
End Sub
```

La opción *Enlace caliente* establece un enlace caliente entre el cliente y el servidor. El procedimiento conducido por el suceso **Click** para este control es el siguiente:

```
Sub EnlaceCaliente_Click ()
  Petición.Visible = FALSE 'ocultar este botón
  Text1.LinkMode = NONE 'romper en enlace
  Text1.LinkMode = HOT 'enlace caliente
End Sub
```

Observe que cuando el tipo de enlace es caliente, el botón *Petición*, que actualiza la caja de texto cuando el enlace es frío, se oculta ya que la actualización es automática.

El botón *Petición* cuando el tipo de enlace es frío, permite actualizar *Text1* con los datos procedentes del servidor. De acuerdo con esto, el procedimiento conducido por el suceso **Click** para el control *Petición* es el siguiente:

```
Sub Petición_Click ()
  'Actualiza la información del cliente
  Text1.LinkRequest
End Sub
```

El botón *Envío* con cualquier tipo de enlace, permite enviar los datos del cliente al servidor. Por lo tanto, el procedimiento conducido por el suceso **Click** para el control *Envío* es el siguiente:

```
Sub Envío_Click ()
  'Envía información al servidor desde el cliente
  Text1.LinkPoke
End Sub
```

Guarde la aplicación y ejecútela. Recuérde que la aplicación que tiene que ejecutar es *Cliente* ya que es ésta quien se encarga de arrancar el servidor. Cuando arranque la aplicación *Cliente*, esperará ver en la pantalla las dos formas, la correspondiente al cliente y la correspondiente al servidor. Si no es así, arrastre la forma correspondiente al cliente y verá que la del servidor se encontraba debajo.

Si lo desea puede crear un fichero CLIENTE.EXE para poder ejecutar esta aplicación bajo Windows. Para crear este fichero, ejecute la orden **Make EXE File** del menú **File**.

Este ejemplo, puede hacerlo extensivo a otros casos. Por ejemplo, el cliente es una aplicación Visual Basic y el servidor es **Excel**. En este caso, cuando se inicie la aplicación correspondiente al cliente, debe iniciarse también la aplicación **Excel**, puesto que el siguiente paso es crear un enlace entre ambas. El procedimiento siguiente conducido por el suceso **Load** se encarga de arrancar la aplicación **Excel** y de crear el enlace entre ambas.

```
Sub Form_Load ()
  'Este procedimiento arranca la aplicación Excel
  'que funciona como servidor
  Dim Id As Integer

  Id = Shell("c:\excel\excel c:\excel\hoja1.xls", 1)
  'La siguiente sentencia asegura que el proceso de
  'cargar Excel finalizará antes de intentar
  'establecer el enlace
  Id = DoEvents()

  Text1.LinkMode = NONE 'romper enlace si existe
  Text1.LinkTopic = "Excel|hoja1.xls"
  Text1.LinkItem = "L1C1"
  Text1.LinkMode = COLD 'enlace frío
End Sub
```

Para un cliente idéntico al anterior, el resto de los procedimientos pueden ser los mismos.

SUCESOS ASOCIADOS CON LOS ENLACES

Los controles que pueden actuar como clientes en una conversación DDE, reconocen los siguientes sucesos: **LinkOpen**, **LinkClose** y **LinkError**.

El suceso **LinkOpen** ocurre cuando un control inicia una conversación DDE con una aplicación servidor. Si el control es *Text1*, el esquema del procedimiento conducido por este suceso es:

```
Sub Text1_LinkOpen (Cancel As Integer)

End Sub
```

El parámetro *Cancel* que se evalúa al final del procedimiento conducido por el suceso **LinkOpen**, indica al cliente si la conversación DDE se ha establecido (*Cancel = 0*) o no (*Cancel <> 0*). Si nosotros ponemos este parámetro a un valor distinto de cero, el enlace se rehusa y si lo dejamos a valor cero entonces es que el enlace se acepta.

El suceso **LinkClose** ocurre cuando el servidor finaliza la conversación DDE que mantiene con el control. Si el control es *Text1*, el esquema del procedimiento conducido por este suceso es:

```
Sub Text1_LinkClose ()

End Sub
```

El suceso **LinkError** ocurre cuando durante una conversación DDE ocurre un error (no nos referimos a un error en el código). Por ejemplo los datos llegan en un formato no reconocido por el cliente. Si el control es *Text1*, el esquema del procedimiento conducido por este suceso es:

```
Sub Text1_LinkError (LinkErr As Integer)

End Sub
```

El parámetro *LinkErr* es el código del error ocurrido.

Cuando una forma está actuando como servidor en una conversación DDE, reconoce los siguientes sucesos: **LinkOpen**, **LinkClose**, **LinkError** y **LinkExecute**.

El suceso **LinkOpen** ocurre cuando una aplicación cliente inicia una conversación DDE con una forma. El esquema del procedimiento conducido por este suceso es:

```
Sub Form_LinkOpen (Cancel As Integer)

End Sub
```

El parámetro *Cancel* que se evalúa al final del procedimiento conducido por el suceso **LinkOpen**, indica si la conversación DDE se ha establecido o no. Si

nosotros ponemos este parámetro a un valor distinto de cero, el enlace se rehusa y si lo dejamos a valor cero entonces es que el enlace se acepta.

El suceso **LinkClose** ocurre cuando el cliente finaliza la conversación DDE que mantiene con el servidor. El esquema del procedimiento conducido por este suceso es:

```
Sub Form_LinkClose ()

End Sub
```

El suceso **LinkError** ocurre cuando durante una conversación DDE ocurre un error (no nos referimos a un error en el código) y la forma está actuando como servidor. El esquema del procedimiento conducido por este suceso es:

```
Sub Form_LinkError (LinkErr As Integer)

End Sub
```

El parámetro *LinkErr* es el código del error ocurrido.

El suceso **LinkExecute** ocurre cuando una aplicación cliente envía una orden a una aplicación servidor para que la ejecute. El esquema del procedimiento conducido por este suceso es:

```
Sub Form_LinkExecute (CmdStr As String, Cancel As Integer)

End Sub
```

El parámetro *CmdStr* es la orden enviada por la aplicación cliente y el parámetro *Cancel* que se evalúa al final del procedimiento conducido por el suceso **LinkExecute**, indica al cliente si la orden fue aceptada (*Cancel = 0*) o no fue aceptada (*Cancel <> 0*).

```
Sub Form_LinkExecute (CmdStr As String, Cancel As Integer)
  Cancel = 0
  Select Case LCase$(CmdStr)
    Case "..."
    ...
    Case Else
      Cancel = -1 'la orden no fue aceptada
  End Select
End Sub
```

ENVIAR ORDENES A OTRAS APLICACIONES

Utilizando el método **LinkExecute**, se pueden enviar a través de un enlace órdenes a otras aplicaciones que soportan DDE. La sintaxis para este método es la siguiente:

Control.**LinkExecute** *ordenS*

Control es el cliente que envía la orden especificada por *orden*. El servidor la recibe y si la reconoce la ejecuta, en otro caso envía un mensaje de error. La sintaxis de una orden depende de la aplicación servidor. Para conocer esta información, consulte la documentación de la aplicación en cada caso. Por ejemplo. Excel para Windows acepta órdenes encerradas entre corchetes.

```
Text1.LinkExecute "[ARCHIVO.CERRAR()]"

Text1.LinkExecute "[BIP][ARCHIVO.CERRAR()][SALIR()]"
```

ACTUALIZAR GRÁFICOS EN UN DDE

Si un cliente inicia una conversación con una caja de texto de una forma que actúa como servidor, cuando la información de ésta cambia. Visual Basic actualiza automáticamente el cliente si se trata de un enlace caliente, o se lo notifica si se trata de un enlace frío.

Sin embargo, Visual Basic no hace lo mismo cuando se trata de una caja de imagen. La razón es que en un enlace caliente, los datos son transferidos desde el servidor al cliente cada vez que hay un cambio en los mismos, lo cual significa que el servidor tendría que enviar todo el gráfico al cliente cada vez que se modificara un pixel. Esto conllevaría un gasto grande de tiempo y de recursos. Por eso, cuando se necesite actualizar el gráfico en el cliente. Visual Basic permite llamar explícitamente desde el servidor al método **LinkSend**. Por ejemplo.

```
Sub Picture1_Paint ()
    'Aquí escriba las operaciones gráficas que desea
    'realizar en el servidor
    Picture1.LinkSend 'envía el gráfico al cliente
End Sub
```

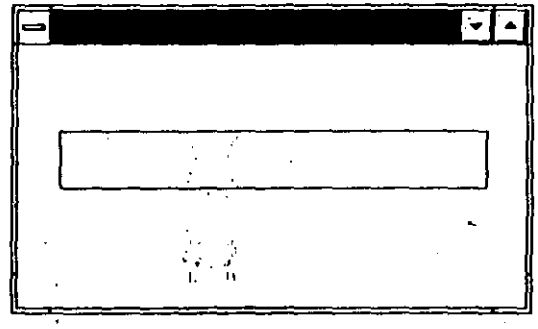

MANIPULACIÓN DE ERRORES

Cuando se ejecuta una aplicación Visual Basic que funciona como cliente o como servidor, se pueden presentar dos clases de errores:

- Errores que ocurren al ejecutarse el código escrito para la aplicación.
- Errores que ocurren durante una conversación DDE (cuando no se ejecuta el código).

La primera clase de errores ya ha sido tratada en capítulos anteriores. Por ejemplo, si una aplicación Visual Basic que funcione como cliente quiere iniciar una conversación con la aplicación **Excel**, lo que hay que hacer previamente es cargar la aplicación que va a funcionar como servidor y después iniciar la conversación. En caso contrario Visual Basic mandará un mensaje de error, de código 282, indicando que no se puede establecer el enlace con el servidor por no estar activa dicha aplicación.

Para ilustrar este ejemplo, inicie una nueva aplicación y cree una forma titulada *Cliente*. Añada a la forma una caja de texto de nombre *Text1*. Guarde la aplicación y la forma.



Cuando el usuario arranque la aplicación, se da el suceso **Load** para el objeto *Form*. De acuerdo con lo enunciado anteriormente, la respuesta a este suceso tiene que ser la siguiente:

- Interceptar los errores que puedan ocurrir al ejecutarse el código escrito de la aplicación cliente.
- Establecer un enlace caliente para DDE entre la aplicación cliente y la aplicación servidor.

- Si ocurre el error 282 (el servidor no está activo), ejecutar la aplicación que va a funcionar como servidor e intentar de nuevo establecer el enlace.
- Si ocurre otro tipo de error, enviar un mensaje al usuario indicándole el código del error ocurrido y finalizar la aplicación.

El código que ejecuta el proceso descrito se indica a continuación.

```
Sub Form_Load ()
  'Este procedimiento arranca la aplicación Excel
  'que funciona como servidor
  Dim Id As Integer
  Const NONE = 0
  Const HOT = 1

  On Error GoTo RutinaDeError

Enlace:
  Text1.LinkMode = NONE 'romper enlace
  Text1.LinkTopic = "Excel|Hoja1"
  Text1.LinkItem = "L1C1"
  Text1.LinkMode = HOT 'enlace caliente

  Exit Sub

RutinaDeError:
  If Err = 282 Then 'no está activa la aplicación
    Id = Shell("c:\excel\excel.exe", 1)
    Resume Enlace
  Else
    MsgBox "Error número: " + Str$(Err)
    End
  End If
End Sub
```

La segunda clase de errores se manipula a través del procedimiento correspondiente conducido por el suceso **LinkError**. Los errores que se pueden dar en una conversación DDE están definidos el fichero CONSTANT.TXT y son los siguientes:

```
' ErrNum (LinkError)
Global Const WRONG_FORMAT = 1
Global Const REQUEST_WITHOUT_INIT = 2
Global Const DDE_WITHOUT_INIT = 3
Global Const ADVISE_WITHOUT_INIT = 4
Global Const POKE_WITHOUT_INIT = 5
Global Const DDE_SERVER_CLOSED = 6
Global Const TOO_MANY_LINKS = 7
Global Const STRING_TOO_LONG = 8
```



```
Global Const INVALID_CONTROL_ARRAY_REFERENCE = 9
Global Const UNEXPECTED_DDE = 10
Global Const OUT_OF_MEMORY = 11
Global Const SERVER_ATTEMPTED_CLIENT_OPERATION = 12
```

Por ejemplo, suponga que tiene una caja de texto llamada *Text1* trabajando como cliente de una conversación DDE. Para manipular los errores que puedan ocurrir durante la conversación DDE, escriba el procedimiento que se presenta a continuación. En primer lugar cargue el fichero CONSTANT.TXT en el módulo global, ya que en él están definidas las constantes referentes a los errores DDE.

```
Sub Text1_LinkError (ErrNum As Integer)
  'Las constantes que recogen los códigos de error
  'están definidas en Global.Bas
  Dim Msg As String

  Select Case ErrNum
    Case OUT_OF_MEMORY
      Msg = "No hay suficiente memoria para ejecutar DDE"
    Case WRONG_FORMAT
      Msg = "Los datos para el DDE no están en un formato correcto"
    Case Else
      Msg = "Error " + Str$(NumError) + " en un DDE"
  End Select
  MsgBox Msg
End Sub
```

Cuando la aplicación VB es el servidor, el procedimiento es exactamente el mismo que cuando la aplicación VB es el cliente, excepto que ahora está asociado a la forma.

```
Sub Form_LinkError (LinkErr As Integer)
  'Las constantes que recogen los códigos de error
  'están definidas en Global.Bas
  Dim Msg As String

  Select Case ErrNum
    Case OUT_OF_MEMORY
      Msg = "No hay suficiente memoria para ejecutar DDE"
    Case WRONG_FORMAT
      Msg = "Los datos para el DDE no están en un formato correcto"
    Case Else
      Msg = "Error " + Str$(NumError) + " en un DDE"
  End Select
  MsgBox Msg
End Sub
```

FUNCIÓN DoEvents

La función **DoEvents** interrumpe un proceso Visual Basic y cede el control al entorno operativo (por ejemplo, Windows), hasta que todos los mensajes que están esperando en la cola de procesos sean ejecutados. Esta función devuelve un valor que se corresponde con el número de formas VB cargadas.

Para entender la importancia de esta función, piense que su aplicación Visual Basic no está sola, sino que está corriendo simultáneamente con otras aplicaciones, lo que implica que todas ellas tienen que compartir los recursos del ordenador. En este sentido, cuando una aplicación Visual Basic está esperando que ocurra un suceso, automáticamente está compartiendo el ordenador con las otras aplicaciones. Sin embargo, cuando uno de sus procedimientos se está ejecutando, Visual Basic no comparte el ordenador con otras aplicaciones. Esto puede originar problemas diversos: por ejemplo, que otra aplicación no pueda responder a una conversación DDE porque no puede coger el control, lo que daría lugar a un mensaje de error. Para evitar esto, puede utilizar la función **DoEvents**.

El siguiente ejemplo, carga la aplicación **Excel**, enlaza el cliente *Text1* con esa aplicación que funciona como servidor y hace una verificación para evitar posibles errores, enviando datos a las celdas de la hoja de cálculo "c:\vb\hoja1.xls", con las que se va a establecer la conversación.

```
Sub Forma_Load ()
  'Este procedimiento arranca la aplicación Excel
  'que funciona como servidor
  Dim Id,'F, C As Integer
  '282: se inicia DDE y la aplicación no responde
  Const DDE_282 = 282
  Const NONE = 0, HOT = 1, LÍNEAS = 6, COLS = 3

  On Error GoTo RutinaDeError

  Id = Shell("c: excel\excel c: vb\hoja1.xls", 1)

Enlace:
  Text1.LinkMode = NONE 'romper enlace
  Text1.LinkTopic = "Excel|c: vb\hoja1.xls"
  Text1.LinkItem = "F1C1"
  Text1.LinkMode = HOT 'enlace caliente

  'Verifica la aplicación
  For L = 1 To LÍNEAS
    For C = 1 To COLS
      Text1.LinkItem = "F" + LTrim$(Str$(L)) -
                       "C" + LTrim$(Str$(C))
```



```
      Text1.Text = Str$(C * L)
      Text1.LinkPoke
      Id = DoEvents()
    Next C, L
    Text1.LinkItem = "F1C1:F" + LTrim$(Str$(LÍNEAS)) -
                     "C" + LTrim$(Str$(COLS))

    Exit Sub

RutinaDeError:
  Select Case Err
    Case DDE_282
      Id = DoEvents()
      Resume Enlace
    Case Else
      MsgBox "Error número: " + Str$(Err)
      End
  End Select
End Sub
```

La sentencia *Id = DoEvents()* incluida en el lazo de verificación, permite que ocurran otros sucesos mientras se ejecuta dicho lazo y la incluida en la rutina de error permite que se establezca el enlace caliente una vez que haya finalizado la carga de la aplicación **Excel**.

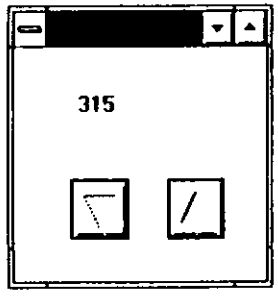
Utilice **DoEvents** para interrumpir un proceso Visual Basic con el fin de que el entorno operativo pueda responder normalmente a otros sucesos que puedan ocurrir periódicamente.

Para clarificar este concepto vamos a exponer un ejemplo que permita realizar un conteo ascendente o descendente mientras el usuario mantenga pulsado el botón del ratón. El conteo se detendrá, cuando el usuario suelte el botón.

Inicie una nueva aplicación y cree una forma titulada *Contador* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *Contador.mak* y la forma con el nombre *Contador.frm*.

Objeto	Propiedad	Valor
Forma	FormName Caption	Form1 Contador
Etiqueta	CtlName Caption	Label1 0

Caja de imagen	CtlName	Picture1
	Autosize	True
	Index	0
	Picture	ARW01DN.ICO
Caja de imagen	CtlName	Picture1
	Autosize	True
	Index	1
	Picture	ARW01UP.ICO



Defina las siguientes variables a nivel de la forma.

```
Dim Pulsado, Contador, Id As Integer
```

El procedimiento conducido por el suceso **MouseDown** para el control *Picture1(Index)*, primero calcula el incremento; si *Index* es 0, *Incremento* vale −1 y si *Index* es 1, *Incremento* vale +1. A continuación pone la variable *Pulsado* a valor **True**. Finalmente, se ejecuta un lazo que realiza el conteo mientras la variable *Pulsado* sea **True**. Cuando el usuario suelte el botón del ratón (suceso **MouseUp**), *Pulsado* tiene que pasar a valor **False** para que se detenga el conteo.

```
Sub Picture1_MouseDown (Index As Integer, Button As
  Integer, Shift As Integer, X As Single, Y As Single)
  If Index Then
      Incremento = 1
  Else
      Incremento = -1
  End If
  Pulsado = -1 'TRUE
  While Pulsado
    Contador = Contador + Incremento
    Label1.Caption = Str$(Contador)
    Id = DoEvents()
  Wend
End Sub
```


Como se ha indicando anteriormente, cuando un procedimiento de una aplicación Visual Basic se está ejecutando, Visual Basic no comparte el ordenador con otras aplicaciones. Esto hace que otros sucesos, por ejemplo **MouseUp**, no sean ejecutados aunque sabemos que el mensaje generado cuando se produce el suceso, se añade a la cola de procesos de Windows para ejecutarse cuando finalice el procedimiento Visual Basic. En el ejemplo anterior, el lazo **While** no finaliza nunca.

Para permitir mirar la cola de procesos y si los hay los ejecute el entorno operativo, hay que llamar a la función **DoEvents**. En el ejemplo, esto lo hacemos cada vez que se ejecuta el lazo **While**, con el fin de que cuando se dé el suceso **MouseUp** se ejecute el procedimiento asociado que se muestra a continuación.

```
Sub Picture1_MouseUp (Index As Integer, Button As
  Integer, Shift As Integer, X As Single, Y As Single)
  Pulsado = 0 'FALSE
End Sub
```

PEGAR VÍNCULOS Y COPIAR

Anteriormente, en este mismo capítulo, hemos visto que en las aplicaciones que soportan enlaces calientes la orden *Pegar Vínculos* (**Paste Link**) permite al usuario crear aplicaciones que funcionan como clientes. Esto exige que la aplicación servidor tenga una orden *Copiar* (**Copy**). Para que las aplicaciones Visual Basic ofrezcan esas mismas características, vamos a estudiar a continuación como añadir la orden *Pegar Vínculos* y la orden *Copiar*.

Para que una orden *Pegar Vínculos* de una aplicación Visual Basic pueda realizar un enlace caliente para que dicha aplicación funcione como cliente, debe obtener la información del portapapeles (**Clipboard**). El método que nos permite obtener esta información es,

Clipboard.GetText ([*formato%*])

Si el argumento *formato%* vale 1 (constante CF_TEXT) el método **GetText** devuelve la cadena de caracteres contenida en el portapapeles y si vale &HBF00 (constante CF_LINK) entonces devuelve una cadena de la forma,

aplicación\tema\elemento de datos

que contiene la información necesaria para realizar el enlace (| es el carácter ASCII 124 y ! es el carácter ASCII 33). El valor de *formato* por defecto es 1.

Las constantes simbólicas que especifican el argumento *formato%* están definidas en el fichero CONSTANT.TXT.

Para que una orden *Copiar* de una aplicación Visual Basic pueda realizar un enlace caliente para que la misma funcione como servidor, debe poner la información *aplicación|tema!elemento de datos* en el portapapeles (**Clipboard**). El método que nos permite poner esta información en el portapapeles es.

Clipboard.SetText *datoS* [*formato%*]

El argumento *datoS* es la cadena que queremos poner en el portapapeles. El argumento *formato%* especifica el formato CF_TEXT (constante de valor 1) o CF_LINK (constante de valor &HBF00); el valor por defecto es 1. La cadena almacenada es de la forma,

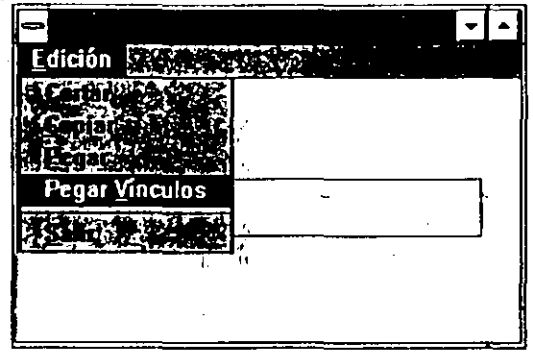
aplicación|tema!elemento de datos

Cuando la aplicación se ejecuta en Visual Basic, *aplicación* es el nombre del proyecto y cuando la aplicación se ejecuta utilizando el fichero ejecutable (.EXE), *aplicación* es el nombre de ese fichero ejecutable. La cadena de caracteres *tema* se corresponde con la propiedad **LinkTopic** de la forma y el *elemento de datos*, es el nombre del control a través del cual se efectúa la conversación. Para tener acceso al nombre de un control, hay almacenar el mismo en la propiedad **Tag**, ya que la propiedad **CtlName** no está disponible en tiempo de ejecución.

Añadir la orden Pegar Vínculos

Inicie una nueva aplicación y cree una forma titulada *Vínculos* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *Vínculos.mak* y la forma con el nombre *Vínculos.frm*.

Objeto	Propiedad	Valor
Menú	Caption CtlName	&Edición MenúEdición
Menú: orden 0	Caption CtlName Index	Co&tar Orden 0
Menú: orden 1	Caption CtlName Index	&Copiar Orden 1
Menú: orden 2	Caption CtlName Index	&Pegar Orden 2
Menú: orden 3	Caption CtlName Index	Pegar &Vínculos Orden 3
Menú: separador	Caption CtlName Index	- guiones 4
Menú: orden 5	Caption CtlName Index	&Salir Orden 5
Forma	FormName Caption	Form1 Vínculos
Caja de texto	CtlName Text Tag	Text1 (nada) Text1



Para ejecutar la orden *Pegar Vínculos* (**Paste Link**) siga los pasos que se especifican a continuación:

1. Obtenga del portapapeles la información para realizar el enlace.
2. Divida la información en las cadenas *aplicación|tema* y *elemento de datos*.
3. Asigne las cadenas anteriores a las propiedades **LinkTopic** y **LinkItem**, respectivamente, del control activo.

Para determinar el control que está activo utilice la propiedad **ActiveControl** del objeto **Screen**.

El procedimiento que da lugar a la orden *Pegar Vínculos* es el siguiente:

```
Sub Orden_Click (Index As Integer)
  'Global.bas contiene CONSTANT.TXT
  Dim Enlace As String, Ad As Integer
  Select Case Index
    Case 0 'Cortar
      'Código que soporte Cortar
    Case 1 'Copiar
      'Código que soporte Copiar
    Case 2 'Pegar
      'Código que soporte Pegar
    Case 3 'Pegar Vínculos
      Enlace = Clipboard.GetText(CF_LINK)
      Ad = InStr(Enlace, "!")
      If Ad Then
        Screen.ActiveControl.LinkMode = NONE
        Screen.ActiveControl.LinkTopic = LeftS(Enlace, Ad - 1)
        Screen.ActiveControl.LinkItem = MidS(Enlace, Ad + 1)
        Screen.ActiveControl.LinkMode = HOT
      ElseIf InStr(Enlace, "|") Then
        'Para los enlaces DDE que no tienen elemento de datos
        Screen.ActiveControl.LinkMode = NONE
        Screen.ActiveControl.LinkTopic = Enlace
        Screen.ActiveControl.LinkItem = " "
        Screen.ActiveControl.LinkMode = HOT
      End If
    Case 5 'Salir
      End
  End Select
End Sub
```

Puede ocurrir que en el portapapeles no haya información válida para realizar un enlace o que el control cliente no sea uno de los requeridos. En este caso, lo que haremos será dejar la orden *Pegar Vínculos* de forma que no se pueda ejecutar (en gris).

```
Sub MenúEdición_Click ()
  'Global.bas contiene CONSTANT.TXT
  Orden(3).Enabled = FALSE 'orden Pegar Vínculos
  If Clipboard.GetFormat(CF_LINK) Then
    If TypeOf Screen.ActiveControl Is PictureBox Then
      If Clipboard.GetFormat(CF_BITMAP) Then
        Orden(3).Enabled = TRUE
      End If
```



```
    ElseIf TypeOf Screen.ActiveControl Is TextBox Then
      If Clipboard.GetFormat(CF_TEXT) Then
        Orden(3).Enabled = TRUE
      End If
    End If
  End If
End Sub
```

En este procedimiento no se ha considerado el control etiqueta porque el usuario no puede seleccionarla; por consiguiente, una etiqueta nunca puede ser un control activo. En adición, el código del procedimiento asegura que el contenido del portapapeles esté en el formato adecuado. Para una caja de imagen, además del formato CF_BITMAP son también válidos CF_METAFILE y CF_DIB.

El método **GetFormat** devuelve un valor –1 si el tipo de formato del contenido del portapapeles es reconocido por Visual Basic; en caso contrario devuelve un 0. Su sintaxis es,

Clipboard.GetFormat (*formato%*)

El argumento *formato%* especifica uno de los siguientes formatos reconocidos por Visual Basic:

Formato	Valor
CF_LINK	&HBF00
CF_TEXT	1
CF_BITMAP	2
CF_METAFILE	3
CF_DIB	8

Las constantes simbólicas que especifican el argumento *formato%* están definidas en el fichero CONSTANT.TXT.

Añadir la orden Copiar

Para añadir la orden *Copiar* de forma que permita que la aplicación Visual Basic pueda funcionar como servidor, escriba en el procedimiento *Orden_Click* anterior el siguiente código:

```
Case 1 'Copiar
  Clipboard.Clear
  'Enlace = aplicación|tomaelelemento_de_datos
  Enlace = "VINCULO" + "|" + Form1.LinkTopic
```

```
Enlace = Enlace + "|" + Screen.ActiveControl.Tag
Clipboard.SetText Enlace, CF_LINK
If TypeOf Screen.ActiveControl Is PictureBox Then
  Clipboard.SetData Screen.ActiveControl.Picture
ElseIf TypeOf Screen.ActiveControl Is TextBox Then
  Clipboard.SetText Screen.ActiveControl.Text
Else
  'Datos para otros controles
End If
```

El método **SetData** cuya sintaxis es.

ClipBoard.SetData (*dato%*, [*formato%*])

permite poner en el portapapeles una imagen *dato%* (propiedad **Image** o **Picture**) utilizando el *formato* 2 (CF_BITMAP), 3 (CF_METAFILE) o 8 (CF_DIB). Por defecto el *formato* es 2.

ENVIAR TECLAS A OTRAS APLICACIONES

Visual Basic permite enviar teclas a una aplicación que esté ejecutándose en el entorno operativo, independientemente de que la aplicación soporte o no DDE. Sin embargo, no se puede enviar teclas a una máquina virtual DOS, esto es, a una sesión DOS ejecutándose en una ventana.

Cuando se envían teclas a otra aplicación, el resultado es el mismo que si el usuario las hubiese pulsado. Por ejemplo, la siguiente sentencia envía las teclas "xyz":

```
SendKeys "xyz"
```

Para enviar teclas especiales, por ejemplo, **F1**, **Tab**, **Enter**, +, se envía el nombre de la tecla encerrado entre llaves. Es indistinto utilizar minúsculas o mayúsculas. Por ejemplo,

```
SendKeys "{F1}{Tab}{Enter}{+}"
```

Las teclas que se envían van automáticamente a la aplicación activa. Quiere esto decir, que si no se activa otra aplicación, las teclas enviadas irán a parar a la misma aplicación. Para activar otra aplicación utilice la sentencia.

AppActivate *nombre$*

El argumento *nombre* se corresponde con el nombre que aparece en la barra de título de la ventana activa de la aplicación.

La sintaxis de la sentencia **SendKeys** es la siguiente:

SendKeys *teclas$*[, *esperar%*]

El argumento *esperar* es una expresión numérica que vale –1 (**True**) o 0 (**False**). Cuando las teclas se envían a otra aplicación y *esperar* vale –1, se procesan las teclas enviadas y después se devuelve el control al procedimiento. Si *esperar* es 0, entonces no se espera a que se procesen las teclas. El argumento *esperar* sólo tiene efecto cuando las teclas se envían a otra aplicación. Si las teclas se envían a la misma aplicación y se necesita esperar a que se procesen, utilice la función **DoEvents**.

CAPÍTULO 14

LLAMADAS A LAS FUNCIONES DE LA API DE WINDOWS

INTRODUCCIÓN

Un programador de aplicaciones Windows además de conocer el entorno de trabajo de Windows debe conocer también su entorno de programación, conocido generalmente como interfaz de programación de aplicaciones de Windows (**Windows Application Programming Interface**, abreviadamente **Windows API**). La característica primaria de la API de Windows son las funciones y los mensajes Windows.

Las funciones Windows son el corazón de las aplicaciones Windows. Hay más de 600 funciones de Windows dispuestas para ser llamadas por ejemplo, desde C o desde Visual Basic. Los mensajes son utilizados por Windows para permitir que dos o más aplicaciones se comuniquen entre sí y con el propio sistema Windows. Se dice que las aplicaciones Windows son conducidas por mensajes o sucesos.

LIBRERÍAS DINÁMICAS

Una librería dinámica (**Dynamic Link Libraries**, abreviadamente **DLLs**) permite que las aplicaciones Windows compartan código y recursos. Una DLL es actualmente un fichero ejecutable que contiene funciones de Windows que pueden ser utilizadas por todas las aplicaciones.

Una de las mejores características de Visual Basic son sus librerías dinámicas. Una DLL es una librería de funciones que Windows lee y ejecuta atendiendo

a las necesidades del programa .EXE que se ejecuta. Lo opuesto a una DLL es una librería estática, donde las funciones necesarias para la ejecución del programa .EXE, son copiadas en el mismo durante el proceso de compilación y enlace que lo genera.

La utilización de librerías dinámicas tiene ventajas. Una ventaja es que como están separadas del programa se pueden actualizar sin tener que modificar los programas que las utilizan. Otra ventaja es el ahorro de memoria principal y de disco ya que como es Windows quien administra la utilización de las DLLs, no existe duplicidad de código cuando varias aplicaciones las utilizan.

También, tiene inconvenientes. Uno de ellos es el tiempo que Windows tiene que emplear en leer las funciones que el programa necesita utilizar de una DLL. Otra desventaja es que cada programa ejecutable necesita que estén presentes las DLLs que utiliza. Cuando se utilizan librerías estáticas, las funciones que el programa necesita se incluyen en el mismo, por lo que ni se pierde tiempo en leerlas ni la librería tiene que estar presente.

Las funciones de la API de Windows están disponibles en las librerías: **Kernel**, **GDI** y **User**. Las DLLs para esas funciones están en los ficheros KERNEL.EXE, GDI.EXE y USER.EXE localizados en el directorio SYSTEM de Windows. Visual Basic puede utilizar DLLs creadas en cualquier lenguaje, con muy pocas excepciones.

Para acceder a las funciones de las librerías dinámicas, Visual Basic utiliza la sentencia **Declare**. Afortunadamente, Microsoft dispone de un fichero llamado WINAPI.TXT que contiene las sentencias **Declare** de casi todas las funciones de la API de Windows, ya que hay unas pocas funciones que Visual Basic no puede utilizar porque trabajan con punteros. Este fichero, hasta ahora, no se suministra con el paquete Visual Basic. No obstante, en el disquete que se adjunta y que contiene los ejercicios desarrollados en este libro, se incluye también este fichero (cortesía de Microsoft).

La mejor forma de utilizar el fichero WINAPI.TXT es cargarlo en el módulo global de la aplicación Visual Basic, no en su totalidad porque es mayor de 64K, sino la parte de interés para la aplicación en curso.

DECLARACIÓN DE UNA FUNCIÓN DE UNA DLL

Para declarar una función de una librería dinámica, tiene que escribir una sentencia **Declare** en el módulo global de la aplicación, o en la sección de declaraciones de la forma o del módulo correspondiente. Por ejemplo,


```
Declare Function lopen Lib "Kernel" Alias "_lopen" (ByVal
    lpPathName As String, ByVal iReadWrite As Integer) As Integer
```

Observe en la declaración las palabras clave **Lib** (Librería) y **ByVal** (por valor). Opcionalmente puede aparecer también la palabra clave **Alias**.

Si la función no retorna un valor, declárela como un procedimiento **Sub** (en lo sucesivo emplearemos el término función para referirnos a una función o a un procedimiento por compatibilidad con otros lenguajes como C). Por ejemplo,

```
Declare Sub CloseWindow Lib "User" (ByVal hWnd As Integer)
```

La cláusula **Lib** indica a Visual Basic la librería donde puede encontrar la función de la API de Windows declarada. Las DLLs del entorno operativo están en "Kernel", "GDI", "User", o en una de las DLLs correspondientes a un **driver** de dispositivo tal como "Sound". Para otras DLLs, el nombre incluye el camino completo. Por ejemplo "c:\windows\system\lzexpand.dll".

Los argumentos de una función pueden ser pasados por valor o por referencia. Para pasar un argumento por valor, escriba la palabra clave **ByVal** delante de la declaración del argumento.

Algunas funciones aceptan más de un tipo de datos para un mismo argumento. Esto se indica declarando el argumento como **Any**. Por ejemplo,

```
Declare Function Catch Lib "Kernel" (lpCatchBuf As Any) As Integer
```

Esta función puede ser llamada con un argumento de tipo **String**, de tipo **Long**, etc.

```
resu% = Catch(arg$)
resu% = Catch(arg&)
```

La cláusula **Alias** indica que la función tiene otro nombre en la librería dinámica (DLL). Esto es útil cuando el nombre de la función coincide con alguna palabra clave de Visual Basic, con alguna variable o constante global, o el nombre de la función en la DLL tiene caracteres no reconocidos por Visual Basic. Por ejemplo,

```
Declare Function lclose Lib "Kernel" Alias "_lclose"
                        (ByVal hFile As Integer) As Integer
```

En este ejemplo *lclose* es el nombre que se utiliza en la aplicación Visual Basic para referirse a la función y *_lclose* es el nombre que tiene la función en la

DLL. Esto se hace así porque en Visual Basic un nombre tiene que comenzar con una letra.

LLAMADA A UNA FUNCIÓN DE UNA DLL

Una vez que se ha declarado una función, puede ser invocada exactamente igual que otras funciones intrínsecas de Visual Basic. Por ejemplo,

```
Declare Function IsIconic Lib "User" (ByVal hWnd As Integer) As Integer

If IsIconic(Form2.hWnd) Then ...
```

Este ejemplo llama a la función **IsIconic** para verificar si la forma *Form2* es un icono.

Visual Basic no puede verificar si los valores de los argumentos que se pasan a una función de una DLL son correctos. Por lo tanto, si se pasan valores incorrectos, la función puede fallar. Quiere esto decir que cuando experimente con funciones de las DLLs, es aconsejable que guarde sus trabajos con frecuencia.

Visual Basic pasa todos los argumentos por referencia. Para pasar un argumento por valor anteponga en la sentencia **Declare** la palabra **ByVal** a la declaración del argumento.

Cuando busque información referente a una función de una DLL, si emplea documentación que utiliza el lenguaje C, recuerde que C pasa todos los argumentos por valor excepto los arrays.

TIPOS DE DATOS EN LAS LLAMADAS

Visual Basic incorpora muchos tipos de datos, de los cuales algunos no son soportados por las funciones de las librerías dinámicas. Por lo tanto, hay que poner una especial atención cuando se llame a estas funciones.

Cadenas de caracteres (Strings)

Las funciones de la mayoría de las DLLs esperan cadenas de caracteres estándar C (ASCIIZ) las cuales finalizan con un carácter nulo (0 ANSI). En este caso, en la declaración de la función anteponga la palabra **ByVal** a la declaración del parámetro. De esta forma Visual Basic interpreta que tiene utilizar el estilo C para pasar la cadena de caracteres. Por la misma razón, una función de una DLL no puede retornar una cadena de caracteres, excepto cuando se haya escrito específicamente para Visual Basic.

Si la función ha sido escrita específicamente para Visual Basic, entonces no necesita utilizar la palabra **ByVal**. En este caso la función podría retornar una cadena de caracteres.

Ya que las cadenas son pasadas por referencia a una función de una DLL, puede suceder que la función modifique la cadena VB. En este caso hay que tener cuidado ya que si la longitud de la variable de caracteres no es suficiente, como la función no puede incrementar dicha longitud, simplemente escribe a continuación del final destruyendo así áreas de memoria. Para evitar este problema declare la cadena de caracteres lo suficientemente grande.

Por ejemplo, la función **GetWindowsDirectory** cuya declaración es,

```
Declare Function GetWindowsDirectory Lib "Kernel"
  (ByVal lpBuffer As String, ByVal nSize As Integer) As Integer
```

devuelve en su primer argumento el camino del directorio Windows. Como medida de seguridad, para llamar a la función declarare el primer argumento como una cadena de longitud 255. Esto es,

```
Dim W As Integer, Camino As String * 255
W = GetWindowsDirectory(Camino, Len(Camino))
```

Otra solución puede ser definir una cadena de longitud variable y rellenarla a nulos. Esto es,

```
Dim W As Integer, Camino As String
Camino = String$(255, 0)
W = GetWindowsDirectory(Camino, Len(Camino))
```

Arrays

Recuerde, a una función VB (**Function** o **Sub**) se la puede pasar como argumento un array íntegro. Por ejemplo,

```
Declare Sub Fx (a(), f%, c%)
...
Dim w(1 To Filas, 1 To Cols)
Call Fx(w(), Filas, Cols)
```

Por el contrario, a una función de una DLL no se la puede pasar un array íntegro de la misma forma que a una función VB. Para pasar un array íntegro a una función de una DLL, se pasa el primer elemento del array por referencia. Esto es así porque los elementos del array están colocados en posiciones sucesivas de memoria. Una función de una DLL a la que se le pasa el primer elemento de un array tiene acceso a todos los elementos del array. Esto no es totalmente cierto para arrays de cadenas de caracteres.

Tipos definidos por el usuario (estructuras)

Algunas funciones de las DLLs aceptan un tipo definido por el usuario como argumento (una estructura). Una estructura hay que pasarla por referencia. Por ejemplo,

```
Type RECT
  Left As Integer
  Top As Integer
  Right As Integer
  Bottom As Integer
End Type

Declare Sub InvertRect Lib "User" (ByVal hDC As Integer, lpRect As RECT)

Dim Posición As RECT
```

La llamada siguiente pasa la estructura *Posición* a la función **InvertRect**.

```
InvertRect hDC, Posición
```

Si la estructura contiene miembros que son cadenas de caracteres, defina éstos de longitud fija.

Punteros nulos

Algunas funciones de las DLLs esperan recibir como argumento un puntero nulo. Para pasar un puntero nulo a una función, declare el argumento de tipo Any y pase la expresión **ByVal 0&**. Por ejemplo,

```
Declare Function FindWindow Lib "User"
  (lpClassName As Any, lpWindowName As Any) As Integer
```

La llamada a esta función será de la forma:


```
hWndExcel = FindWindow(ByVal 0&, ByVal "Microsoft Excel")
```

El echo de utilizar **ByVal** para pasar la cadena de caracteres es, como hemos expuesto anteriormente, para que Visual Basic interprete que tiene que utilizar el estilo C.

Handles

Un **handle** es un valor entero único definido por el entorno operativo y utilizado para referirse a objetos tales como formas y controles; un **handle** es un número de identificación (ID). Por ejemplo, **hWnd** (handle to Window) es un **handle** para referirse a una ventana. **hDC** (handle to Device Context) es un **handle** para referirse al contexto de dispositivo de un objeto. Las funciones de las librerías dinámicas utilizan mucho este tipo de parámetro. Cuando una función de una DLL espera recibir como argumento un **handle** declárelo de tipo **ByVal Integer**. Por ejemplo,

```
Declare Function IsIconic Lib "User" (ByVal hWnd As Integer) As Integer

If IsIconic(Form2.hWnd) Then ...
```

El entorno operativo asigna un **handle** a cada forma de una aplicación VB, para posteriormente identificarlas. La propiedad **hWnd** de una forma, permite acceder a este **handle**. Por ejemplo,

```
Sub Command1_Click ()
  If IsIconic(Form2.hWnd) Then
    MsgBox "Form2 es un icono"
  End If
End Sub
```

Por contexto de dispositivo se entiende un conjunto de atributos (color de fondo, tipo de letra, espaciado entre caracteres, posición actual de la pluma, paleta de colores, etc.) que determinan la localización y la apariencia de un objeto. Una aplicación no puede acceder al contexto de dispositivo directamente pero si lo puede hacer indirectamente a través de la propiedad **hDC** del objeto (forma, caja de imagen, u objeto **Printer**). Por ejemplo,

```
'Declaración de la función Windows
Declare Sub FloodFill Lib "GDI" (ByVal hDC As Integer, ByVal X
        As Integer, ByVal Y As Integer, ByVal Color As Long)

Sub Form_Click ()
  ScaleMode = 3      ' Windows dibuja en pixel
```

```
    ForeColor = 0      ' Se dibuja en negro
    Line (100, 50)-(300, 50)      ' Dibuja un triángulo
    Line -(200, 200)
    Line -(100, 50)
    FillStyle = 0      ' Rellenar con color sólido
    FillColor = RGB(128, 128, 255)      ' Color
    'Invocar a la función para colorear el triángulo
    FloodFill hDC, 200, 100, ForeColor
End Sub
```

Propiedades

Las propiedades deben ser pasadas por valor. Por ejemplo,

```
Declare Function GetDeviceCaps Lib "GDI" (ByVal hDC As
          Integer, ByVal nIndex As Integer) As Integer

Sub Form_Click ()
  Dim H As Integer, V As Integer
  H = 8: V = 10
  Print "Resolución de la pantalla:"
  Print GetDeviceCaps(hDC, H); "x"; GetDeviceCaps(hDC, V)
  Print "Resolución de la impresora:"
  Print GetDeviceCaps(Printer.hDC, H); "x";
        GetDeviceCaps(Printer.hDC, V)
End Sub
```

Este ejemplo obtiene la resolución de la pantalla y de la impresora actualmente seleccionada. Para ello se pasa a la función la propiedad **hDC**.

Para pasar una propiedad por referencia se debe de utilizar una variable intermedia. Todas las cadenas son pasadas por referencia; por lo tanto, para pasar una propiedad que sea una cadena de caracteres a una función de una DLL, primero se asigna la propiedad a una variable de caracteres y después se pasa la variable. Por ejemplo, para almacenar el camino del directorio Windows en la propiedad **Path** de la lista de ficheros *File1*, escriba:

```
Declare Function GetWindowsDirectory Lib "Kernel" (ByVal lpBuffer
                  As String, ByVal nSize As Integer) As Integer

Dim W As Integer, Camino As String
Camino = String$(255, 1)
W = GetWindowsDirectory(Camino, Len(Camino))
File1.Path = Camino
```

Para pasar propiedades numéricas por referencia, utilice la misma técnica.

Formas y controles

No se puede pasar una forma o un control a una función de una DLL a no ser que la función haya sido escrita específicamente para Visual Basic; tampoco se puede pasar cualquiera de los objetos especiales (Screen, Clipboard, Printer, o Debug).

TIPOS DE WINDOWS

El fichero WINDOWS.H define nuevos nombres para algunos de los tipos estándar. La siguiente tabla presenta estos tipos con sus equivalentes en C y en Visual Basic. Esto le facilitará la utilización de documentación para programación en Windows, escrita bajo el lenguaje C.

Windows	C	Visual Basic	Definición
BOOL	int	ByVal Integer	Entero
BYTE	unsigned char		
WORD	unsigned int	ByVal Integer	Entero
DWORD	unsigned long	ByVal Long	Entero largo
LPSTR	char far *	ByVal String	Cadena de caracteres
ATOM	WORD	ByVal Integer	Entero
HANDLE	WORD	ByVal Integer	Entero
HWND	HANDLE	ByVal Integer	Entero
HICON	HANDLE	ByVal Integer	Entero
HDC	HANDLE	ByVal Integer	Entero
HMENU	HANDLE	ByVal Integer	Entero
HBITMAP	HANDLE	ByVal Integer	Entero
COLORREF	DWORD	ByVal Long	Entero largo

Cómo pasar un valor a un parámetro de tipo BYTE

Algunas funciones de las DLLs aceptan un valor de tipo BYTE. Como este tipo de datos no está definido en Visual Basic, se debe pasar un dato de tipo **Integer**. Por ejemplo, la función **GetTempFileName** cuya sintaxis es,

```
int GetTempFileName(BYTE cDriveLetter,
                    LPSTR lpPrefixString,
                    WORD wUnique,
                    LPSTR lpTempFileName)
```

crea un nombre único para un fichero temporal que la aplicación puede utilizar para almacenar datos.

La declaración de esta función en Visual Basic, sería de la forma siguiente:

```
Declare Function GetTempFileName Lib "Kernel" (
            ByVal cDriveLetter As Integer,
            ByVal lpPrefixString As String,
            ByVal wUnique As Integer,
            ByVal lpTempFileName As String) As Integer
```

La razón es que el tipo mas pequeño de datos que se pone en la pila de un microprocesador 80x86 es un palabra. Por lo tanto, el tipo BYTE igual que el **Integer** requiere una palabra de memoria para almacenarse en la pila, lo que nos conduce a resultados idénticos.

MENSAJES

La característica primaria de la API de Windows son las funciones y los mensajes Windows. Haciendo uso de los mensajes de Windows se puede obtener información de un control, que no se puede obtener utilizando sólo las características propias de Visual Basic. Para utilizar las funciones y los mensajes Windows, es necesario conocer de cada una/o de ellas/os su finalidad, sintaxis, valor que devuelve y parámetros que utiliza. Encontrará una amplia información tanto de las funciones como de los mensajes Windows en "Microsoft Windows SDK".

Un mensaje consta de tres partes:

- Número de mensaje, identificado por el nombre del mensaje.
- Un parámetro **word** (16 bits), referenciado por *wParam*.
- Un parámetro **long** (32 bits), referenciado por *lParam*.

Por ejemplo, EM_GETLINE copia una línea de un objeto multilínea a un área de memoria (**buffer**). El parámetro *wParam* define el número de línea; la primera línea es la 0. El parámetro *lParam* apunta al área de memoria donde se almacena la línea. El número máximo de caracteres que se pueden copiar se pone en la primera palabra del **buffer**.

La sintaxis para enviar un mensaje es la siguiente:

```
Declare Function SendMessage Lib "User" (
                                ByVal hWnd As Integer,
                                ByVal wMsg As Integer,
                                ByVal wParam As Integer,
                                lParam As Any) As Long
```


El parámetro *hWnd* es el **handle** de la ventana que recibe el mensaje y el parámetro *wMsg* es el mensaje que se envía. Los parámetros *wParam* y *lParam* contienen información adicional al mensaje que encontrará en la descripción del propio mensaje. Por ejemplo,

```
Pos& = SendMessage(hWnd, EM_GETLINE,
                   CINT(NumLínea), Buffer$)
```

Cada uno de estos mensajes se encuentra definido en un fichero escrito en lenguaje C, del SDK de Windows (Software Development Kit) denominado WINDOWS.H y las declaraciones equivalentes en Visual Basic, se encuentran en el fichero WINAPI.TXT suministrado en el disquete que se adjunta. Por ejemplo,

```
Global Const WM_USER = &H400
Global Const EM_GETLINE = WM_USER+20
```

EDITOR DE TEXTOS

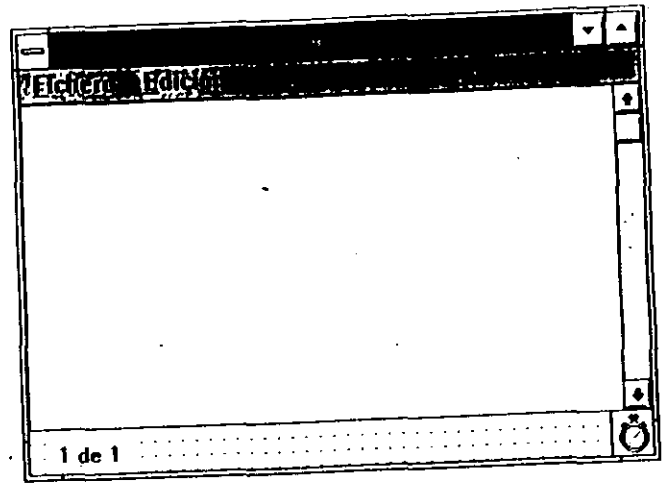
El siguiente ejemplo utiliza una caja de texto multilínea para implementar un editor de textos y para demostrar la utilización de la función **SendMessage** y de varios mensajes de la API de Windows.

El editor realizará las siguientes operaciones:

- Cuando el usuario escriba líneas de texto, se indicará en la línea de estado (línea del fondo) el número de la línea sobre la que está el cursor y el número total de líneas del texto.
- Tendrá un menú *Fichero* con las órdenes, *Nuevo*, *Abrir*, *Guardar* y *Salir*. Con respecto a las dos primeras órdenes, no implementaremos su función pero si visualizarán el mensaje "¿Quiere salvar los cambios?" cuando alguna de ellas se ejecute y el texto haya cambiado. Este mensaje no aparecerá si previamente se ha ejecutado la orden *Guardar*. La orden *Salir* estará operativa y también detectará si el texto ha cambiado y no se ha guardado, con el fin de notificar al usuario si desea guardar los cambios antes de salir.
- Tendrá un menú *Edición* con las órdenes, *Deshacer*, *Cortar*, *Copiar*, *Pegar* y *Borrar*. Todas estas órdenes estarán operativas.

Inicie una nueva aplicación y cree una forma titulada *Editor* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *Editor.mak* y la forma con el nombre *Editor.frm*.

Objeto	Propiedad	Valor
Menú Fichero	Caption CtlName	&Fichero MenúFichero
Fichero: orden 0	Caption CtlName	&Nuevo NuevoFichero
Fichero: orden 1	Caption CtlName	&Abrir AbrirFichero
Fichero: orden 2	Caption CtlName	&Guardar GuardarFichero
Fichero: separador	Caption CtlName	- guiones1
Fichero: orden 3	Caption CtlName	&Salir SalirFichero
Menú Edición	Caption CtlName	&Edición MenúEdición
Edición: orden 0	Caption CtlName	&Deshacer DeshacerEdición
Edición: separador	Caption CtlName	- guiones2
Edición: orden 1	Caption CtlName	Cor&tar CortarEdición
Edición: orden 2	Caption CtlName	&Copiar CopiarEdición
Edición: orden 3	Caption CtlName	&Pegar PegarEdición
Edición: orden 4	Caption CtlName	&Borrar BorrarEdición
Forma	FormName Caption	Form1 Editor
Caja de texto	CtlName Text Multiline ScrollBars	Text1 (nada) True 2 - Vertical
Etiqueta	CtlName Caption AutoSize	Label1 1 de 1 True
Temporizador	CtlName Interval Enabled	Timer1 350 True

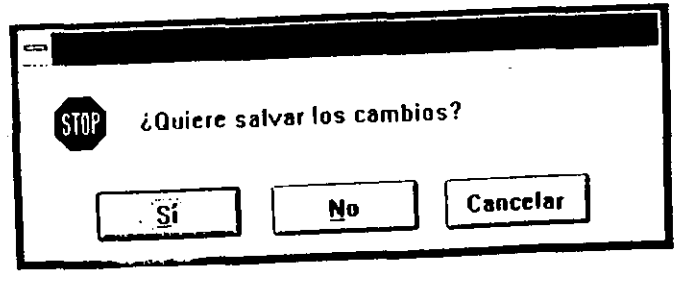


Realice las siguientes definiciones a nivel de la forma.

```
DefInt A-Z
Const CANCELAR = 2, SI = 6, CTRL = 2
```

La sentencia *DefInt A-Z* indica que por defecto todas las variables son enteras. Las constantes *CANCELAR* y *SI* se corresponden con los valores devueltos por la función **MsgBox**.

Cuando el usuario ejecute la orden *Salir*, el procedimiento asociado llama a la función *TextoModificado* para verificar si se ha modificado el texto. Si el texto ha sido modificado, el procedimiento *Mensaje* invoca a la función *MensajeGuardar* que visualiza una ventana con un título que dice "El texto ha sido modificado", una pregunta dirigida al usuario que dice "¿Quiere salvar los cambios?" y tres botones, *Cancelar*, *Si* y *No*, para que el usuario responda a la pregunta formulada. La variable *Botón* almacena esta respuesta (6 = *Si*, 7 = *No* y 2 = *Cancelar*). Si el texto no ha sido modificado, simplemente se finaliza la aplicación.



Escriba el procedimiento *SalirFichero* que se muestra a continuación.

```
Sub SalirFichero_Click ()
  If TextoModificado() Then
    Mensaje
  End If
  End
End Sub
```

Abra la ventana de código e inicie un nuevo procedimiento denominado *Mensaje* (orden **New Procedure** del menú **Code**) y escriba el código que se muestra a continuación.

```
Sub Mensaje ()
  Botón = MensajeGuardar()
  Select Case Botón
    Case CANCELAR
      Exit Sub
    Case SI
      'Aquí, guarde el fichero
  End Select
End Sub
```

Las funciones *TextoModificado* y *MensajeGuardar* se detallan un poco más adelante, en el módulo *Editor.bas*.

Cuando el usuario ejecute la orden *Guardar*, el procedimiento asociado guardará el texto y pondrá el indicador de modificado a cero (**False** - no modificado) para lo cual invocará al procedimiento *NoModificado* que se presenta un poco más adelante, en el módulo *Editor.bas*.

```
Sub GuardarFichero_Click ()
  'Aquí, guarde el fichero
  NoModificado
End Sub
```

Cuando el usuario ejecute la orden *Nuevo*, el procedimiento asociado verificará si se ha modificado el texto actualmente en proceso, en cuyo caso permitirá guardarlo, y a continuación limpiará la ventana de texto y pondrá el indicador de modificado a cero (**False** - no modificado).

```
Sub NuevoFichero_Click ()
  If TextoModificado() Then
    Mensaje
  End If
  Texti.Text = ""
  NoModificado
End Sub
```


Cuando el usuario ejecute la orden *Abrir*, el procedimiento asociado verificará si se ha modificado el texto actualmente en proceso, en cuyo caso permitirá guardarlo, y a continuación cargará en la ventana de texto el fichero elegido de la lista (esta operación no se implementa) y pondrá el indicador de modificado a cero (**False** - no modificado).

```
Sub AbrirFichero_Click ()
  If TextoModificado() Then
    Mensaje
  End If
  'Código para elegir de la lista el fichero que desea abrir
  Texti.Text = "Se visualizará el texto del fichero elegido"
  NoModificado
End Sub
```

Suponga ahora que el usuario sale de la aplicación ejecutando la orden *Cerrar* del menú de control de la ventana. En este caso se da el suceso **Unload** y la forma se descarga. Por lo tanto, el procedimiento conducido por este suceso tiene que ejecutar las mismas acciones que el procedimiento *SalirFichero_Click* con la diferencia de que aquí no se necesita una sentencia **End**.

```
Sub Form_Unload (Cancel As Integer)
  If TextoModificado() Then
    Botón = MensajeGuardar()
    Select Case Botón
      Case CANCELAR
        Cancel = -1
        Exit Sub
      Case SI
        'Aquí, guarde el fichero
    End Select
  End If
End Sub
```

El parámetro *Cancel* de este procedimiento es verificado automáticamente por el sistema al finalizar la ejecución del mismo. Si su valor es cero la forma se descarga; si su valor es distinto de cero la forma no se descarga. Entonces, si el usuario intenta cerrar la aplicación a través del menú de control y el texto ha sido modificado, igual que en procedimientos anteriores se visualiza una caja de diálogo advirtiéndole de este hecho; si pulsa el botón *Cancelar* de esta caja, la forma no debe descargarse para lo cual hay que poner el parámetro *Cancel* a un valor distinto de cero.

Para reflejar en la línea de estado el número de línea donde actualmente se encuentra el cursor (*LActual*) y el número total de líneas (*TLíneas*) se utiliza un temporizador cuyo procedimiento asociado actualiza cada 350 segundos

(propiedad **interval**), las variables que contienen estos valores. Las variables *LActualAnt* y *TLíneasAnt* contienen los valores anteriores a los actuales.

Escriba el siguiente procedimiento conducido por el suceso **Timer**. La función *LíneaActual* devuelve el número de línea donde actualmente se encuentra el cursor y la función *TotalLíneas* el número total de líneas. Ambas funciones se detallan un poco más adelante, en el módulo *Editor.bas*.

```
Sub Timer1_Timer ()
  LActual = LíneaActual()
  TLíneas = TotalLíneas()
  If LActual <> LActualAnt Or TLíneas <> TLíneasAnt Then
    Label1.Caption = Str$(LActual) + " de" + Str$(TLíneas)
    LActualAnt = LActual
    TLíneasAnt = TLíneas
  End If
End Sub
```

A continuación vamos a escribir los procedimientos para las órdenes correspondientes al menú *Edición*.

```
Sub DeshacerEdición_Click ()
  Deshacer
End Sub

Sub CortarEdición_Click ()
  Cortar
End Sub

Sub CopiarEdición_Click ()
  Copiar
End Sub

Sub PegarEdición_Click ()
  Pegar
End Sub

Sub BorrarEdición_Click ()
  Borrar
End Sub
```

Estos procedimientos llaman a otros tantos procedimientos definidos en el módulo *Editor.bas*, que tras enviar un mensaje a Windows realizan la operación a la que hacen referencia.

Para completar la aplicación vamos a desarrollar un procedimiento que permita borrar la línea sobre la que está el cursor, cuando el usuario pulse las teclas **Ctrl+Y**. Para ello, hay que tener en cuenta que la línea 1 de nuestro editor es la 0 para Windows, valor devuelto por el mensaje EM_LINEFROMCHAR.

```
Sub Text1_KeyDown (KeyCode As Integer, Shift As Integer)
  If Shift = CTRL And KeyCode = Asc("Y") Then
    LActual = LíneaActual() - 1 'para Windows una menos
    Text1.SelStart = CaracteresAntes(LActual)
    LongLínea = CaracteresAntes(LActual + 1) - Text1.SelStart
    Text1.SelLength = LongLínea
    Text1.SelText = ""
  End If
End Sub
```

Para finalizar la aplicación, cree un nuevo módulo llamado *Editor.bas* y escriba las siguientes declaraciones, definiciones, funciones y procedimientos referenciadas/os en los procedimientos hasta ahora expuestos. Simplemente por claridad en el diseño, el autor ha preferido incluir todos los procedimientos y funciones que llaman a funciones de la API de Windows en un módulo aparte. Al final podrá comprobar que es posible prescindir de este módulo y por lo tanto reducir el número total de procedimientos desarrollados.

```
DefInt A-Z
Declare Function SendMessage Lib "User" (
                                ByVal hWnd As Integer,
                                ByVal wMsg As Integer,
                                ByVal wParam As Integer,
                                lParam As Any) As Long

Declare Function GetFocus Lib "User" () As Integer

'Constantes correspondientes a los mensajes que serán enviados
Const WM_USER = 1024
Const WM_CUT = 768
Const WM_COPY = 769
Const WM_PASTE = 770
Const WM_CLEAR = 771

Const EM_GETMODIFY = WM_USER + 8
Const EM_SETMODIFY = WM_USER + 9
Const EM_LINEINDEX = WM_USER + 11
Const EM_LINEFROMCHAR = WM_USER + 25
Const EM_UNDO = WM_USER + 23
Const EM_GETLINECOUNT = WM_USER + 10

Function MensajeGuardar ()
  TítuloS = "El fichero ha sido modificado"
```

```
    MsgS = "¿Quiere salvar los cambios?"
    MensajeGuardar = MsgBox(MsgS, 19, TítuloS)
End Function

Function TextoModificado ()
    hWnd = GetFocus()
    TextoModificado = SendMessage(hWnd, EM_GETMODIFY, 0, 0&) * -1
End Function

Sub NoModificado ()
    hWnd = GetFocus()
    vr = SendMessage(hWnd, EM_SETMODIFY, 0, 0&) * -1
End Sub

Function LíneaActual ()
    hWnd = GetFocus()
    LíneaActual = SendMessage(hWnd, EM_LINEFROMCHAR, -1, 0&) + 1
End Function

Function TotalLíneas ()
    hWnd = GetFocus()
    TotalLíneas = SendMessage(hWnd, EM_GETLINECOUNT, 0, 0&)
End Function

Sub Deshacer ()
    hWnd = GetFocus()
    vr = SendMessage(hWnd, EM_UNDO, 0, 0&)
End Sub

Sub Cortar ()
    hWnd = GetFocus()
    vr = SendMessage(hWnd, WM_CUT, 0, 0&)
End Sub

Sub Copiar ()
    hWnd = GetFocus()
    vr = SendMessage(hWnd, WM_COPY, 0, 0&)
End Sub

Sub Pegar ()
    hWnd = GetFocus()
    vr = SendMessage(hWnd, WM_PASTE, 0, 0&)
End Sub

Sub Borrar ()
    hWnd = GetFocus()
    vr = SendMessage(hWnd, WM_CLEAR, 0, 0&)
End Sub
```



```
Function CaracteresAntes (LActual As Integer)
    hWnd = GetFocus()
    CaracteresAntes = SendMessage(hWnd, EM_LINEINDEX, LActual, 0&)
End Function
```

En este módulo aparecen en primer lugar las sentencias **Declare** para las funciones de la API de Windows **SendMessage** y **GetFocus**.

La utilización de la función **SendMessage** depende de la propiedad **hWnd** del objeto al cual se quiere enviar el mensaje. Desafortunadamente, esta propiedad sólo la tienen las formas. No obstante, existen varias maneras de encontrar el **handle** (hWnd) de un control utilizando las funciones de la API de Windows. La más sencilla es utilizar la función **GetFocus**. Esta función devuelve el **hWnd** del control que actualmente posee el foco.

Los valores de *WM_USER* y superiores, hasta 32767, pueden ser utilizados por una aplicación para definir mensajes. Observe que Windows define un mensaje como un número *WM_USER+valor*. Si usted necesita definir un mensaje de este modo, elija un *valor* que no se solape con los definidos en el fichero WINDOWS.H (o WINAPI.TXT); por ejemplo, por encima de 200. La siguiente tabla especifica los mensaje utilizados en esta aplicación.

Mensaje	Significado
WM_CUT	Mueve el texto seleccionado en un control al portapapeles.
WM_COPY	Copia el texto seleccionado en un control en el portapapeles.
WM_PASTE	Copia el texto del portapapeles en un control.
WM_CLEAR	Borra el texto seleccionado en un control sin copiarlo en el portapapeles.
EM_UNDO	Deshace el último cambio.
EM_GETMODIFY	Determina si se ha modificado el texto del control.
EM_SETMODIFY	Activa (*wParam* = **True**) o desactiva (*wParam* = **False**) el indicador de modificado del control.
EM_LINEINDEX	Devuelve la posición que ocupa en el texto el primer carácter de la línea especificada. Este valor se corresponde con el número de caracteres que hay antes del principio de la línea.
EM_LINEFROMCHAR	Devuelve el número de línea correspondiente al texto seleccionado (*wParam* = -1) o la posición en el texto de un determinado carácter (*wParam* > 0).
EM_GETLINECOUNT	Determina el número de líneas de texto que hay en un control multilínea.

Lógicamente, nos estamos refiriendo a un control que permite la edición. En nuestro caso se trata de una caja de texto multilínea.

BORRAR UNA LISTA

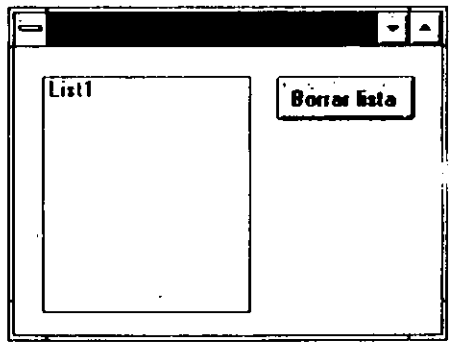
El siguiente ejemplo muestra como borrar rápidamente una lista. Hasta ahora este proceso lo hemos realizado borrando un elemento cada vez. Esto es,

```
Do While List1.ListCount > 0
  List1.RemoveItem 0
Loop
```

Para borrar todas las entradas de una lista de una sola vez, simplemente hay que enviar a la misma el mensaje LB_RESETCONTENT de Windows.

Inicie una nueva aplicación y cree una forma titulada *Lista* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *Lista.mak* y la forma con el nombre *Lista.frm*.

Objeto	Propiedad	Valor
Lista	CtlName Sorted	List1 True
Botón	Caption CtlName	Borrar lista BorrarLista



Escriba la siguiente sentencia en la sección de declaraciones de la forma.

```
DefInt A-Z
```


Para probar la aplicación necesitamos añadir algún elemento a la lista. Este proceso puede realizarse cuando se inicia la aplicación. Para ello, escriba el procedimiento *Form_Load* que se presenta a continuación.

```
Sub Form_Load ()
  For I = 1 To 15
    List1.AddItem "Entrada_" + Format$(I, "00")
  Next I
  List1.ListIndex = 0 'seleccionar primera entrada
End Sub
```

Cuando el usuario pulse el botón *Borrar lista* todas las entradas de la lista se borrarán. Para realizar esta tarea el procedimiento asociado con este botón, invoca al procedimiento *Borrar_Lista* y le pasa como argumento el control *List1*. Este procedimiento, que se detalla a continuación en el módulo *Lista.bas*, coge el **handle** de *List1* para a continuación enviarle el mensaje correspondiente.

```
Sub BorrarLista_Click ()
  Borrar_Lista List1
End Sub
```

Para finalizar la aplicación, cree un nuevo módulo llamado *Lista.bas* y escriba las siguientes declaraciones, definiciones, funciones y procedimientos referenciadas/os en los procedimientos hasta ahora expuestos.

```
DefInt A-Z

Declare Function SendMessage Lib "User" (
                             ByVal hWnd As Integer,
                             ByVal wMsg As Integer,
                             ByVal wParam As Integer,
                             lParam As Any) As Long

Declare Function GetFocus Lib "User" () As Integer

Declare Function SetFocusAPI Lib "User" Alias
        "SetFocus" (ByVal hWnd As Integer) As Integer

Const WM_USER = 1024
Const LB_RESETCONTENT = WM_USER + 5
```

Para enviar un mensaje a un control, primero hay que salvar el **handle** del control que actualmente está enfocado, después se enfoca el control al que se quiere enviar el mensaje y se recupera su **handle** y por último, se vuelve a enfocar el control original. Esto es,

```
Function hWndControl (Ctrl As Control)
  hWnd = GetFocus() 'handle del control que está enfocado
  Ctrl.SetFocus 'enfocar nuestro control
  hWndControl = GetFocus() 'handle de nuestro control
  vr = SetFocusAPI(hWnd) 'enfocar el control original
End Function
```

El mensaje LB_RESETCONTENT borra todos los elementos de una lista y libera la memoria asociada con ellos. La sintaxis para enviar este mensaje es la siguiente:

```
vr = SendMessage(hWnd%, Msg%, wParam%, lParam)
```

donde *hWnd* identifica al control que va a recibir el mensaje. *Msg* es el mensaje a enviar (&H405) y *wParam* y *lParam* no se utilizan (valor cero).

```
Sub Borrar_Lista (Ctrl As Control)
  hWnd = hWndControl(Ctrl) 'recupera el handle de Ctrl
  vr = SendMessage(hWnd, LB_RESETCONTENT, 0, 0)
End Sub
```

PALABRA DE PASO

Un control tiene varios estilos de edición: por ejemplo, texto ajustado a la izquierda, texto centrado, desplazamiento vertical automático, texto en mayúsculas, palabra de paso, etc. El estilo utilizado para introducir una *palabra de paso* es que cualquier carácter que se escriba se visualice como un asterisco (*). Visual Basic no tiene ninguna función para que cuando se escriba un carácter en un control, se visualice un asterisco en su lugar. Para conseguir ésto, hay que utilizar las siguientes funciones de la API de Windows: **GetFocus**, **GetWindowLong**, **SetWindowLong** y **SendMessage**.

Para crear una caja de texto con estilo *palabra de paso* hay que realizar los pasos siguientes:

1. Utilizando la función **GetFocus** se toma el **handle** de la caja de texto.

   ```
   hWnd% = GetFocus()
   ```

2. A continuación, utilizando el argumento GWL_STYLE con la función **GetWindowLong** se recupera el estilo de la caja de texto.

   ```
   Estilo& = GetWindowLong(hWnd%, GWL_STYLE)
   ```

3. Después, utilizando la función **SetWindowLong** se pone el nuevo estilo a la le texto.


```
Estilo& = Estilo& Or ES_PASSWORD
Estilo& = SetWindowLong(hWnd%, GWL_STYLE, Estilo&)
```

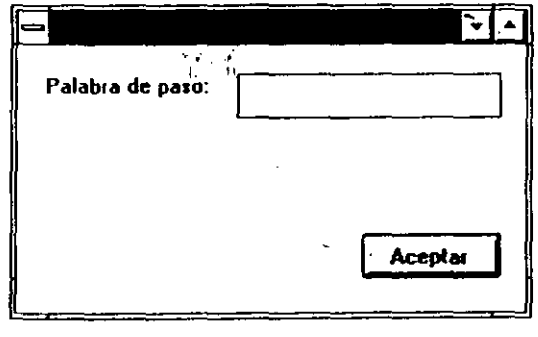
4. Por último, enviando el mensaje ES_SETPASSWORDCHAR se informa al control, del carácter que se va a utilizar como máscara para la palabra de paso. El carácter utilizado normalmente es un asterisco (*).

   ```
   MáscaraPPaso% = Asc("*")
   vr = SendMessage(hWnd%, ES_SETPASSWORDCHAR, MáscaraPPaso%, 0&)
   ```

El siguiente ejemplo muestra como utilizar una caja de texto para introducir una palabra de paso.

Inicie una nueva aplicación y cree una forma titulada *Form1* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *ppaso.mak* y la forma con el nombre *ppaso.frm*.

Objeto	Propiedad	Valor
Etiqueta	Caption TabIndex	Palabra de paso: 0
Caja de texto	CtlName Text TabIndex	PPaso (nada) 1
Botón	Caption CtlName Default TabIndex	Aceptar Aceptar True 2



Cuando el usuario ejecute la aplicación, la forma *Form1* se cargará y la caja de texto *PPaso* quedará enfocada. Esto hace que se dé el suceso **GotFocus** para este control y por lo tanto, que se ejecute el procedimiento *PPaso_GotFocus*.

En tiempo de ejecución el cursor lo recibe el control que tenga el valor de **TabIndex** más bajo de aquellos que pueden recibirlo (los menús y los temporizadores no tienen propiedad **TabIndex** y en tiempo de ejecución, no pueden recibir el cursor los controles invisibles o desactivados, las etiquetas y los marcos).

El procedimiento *PPaso_GotFocus*, utilizando la variable estática *SeEjecutó*, permitirá que se ejecute una sola vez el procedimiento *EstiloPPaso* que pone a la caja de texto *PPaso* el estilo de edición de *palabra de paso*.

```
Sub PPaso_GotFocus ()
  Static SeEjecutó As Integer
  If Not SeEjecutó Then
    SeEjecutó = Not SeEjecutó 'igual a True
    EstiloPPaso Asc("*")
  End If
End Sub
```

El procedimiento *EstiloPPaso* se escribe a continuación en el módulo *ppaso.bas*.

Cuando el usuario haya escrito la palabra de paso, pulsará **Enter** o el botón *Aceptar*. Esto hace que se ejecute el procedimiento *Aceptar_Click* que comprueba si es correcta la palabra de paso; si no es correcta se permitirá al usuario repetir la entrada hasta un máximo de tres veces.

```
Sub Aceptar_Click ()
  Dim Clave As String * 9
  Static Cuenta As Integer
  Clave = PPaso.Text
  If UCase$(Clave) <> "CANIER  " Then
    MsgBox "La palabra de paso no es correcta", 16
    Cuenta = Cuenta + 1
    If Cuenta = 3 Then End
    PPaso.Text = ""
    PPaso.SetFocus
  Else
    'Continúe el proceso
  End If
End Sub
```

Para finalizar la aplicación, cree un nuevo módulo llamado *ppaso.bas* y escriba las siguientes declaraciones, definiciones, funciones y procedimientos referenciadas/os en los procedimientos hasta ahora expuestos.


```
Declare Function SendMessage Lib "User" (
                             ByVal hWnd As Integer,
                             ByVal wMsg As Integer,
                             ByVal wParam As Integer,
                             lParam As Any) As Long

Declare Function GetFocus Lib "User" () As Integer

Declare Function GetWindowLong Lib "User" (
                        ByVal hWnd As Integer,
                        ByVal nIndex As Integer) As Long

Declare Function SetWindowLong Lib "User" (
                        ByVal hWnd As Integer,
                        ByVal nIndex As Integer,
                        ByVal dwNewLong As Long) As Long

Const WM_USER = 1024
Const EM_SETPASSWORDCHAR = WM_USER + 28
Const ES_PASSWORD = 32
Const GWL_STYLE = -16

Sub EstiloPPaso (Máscara As Long)
  Dim hWnd As Integer, Estilo As Long, vr As Long
  hWnd = GetFocus
  Estilo = GetWindowLong(hWnd, GWL_STYLE)
  Estilo = Estilo Or ES_PASSWORD
  Estilo = SetWindowLong(hWnd, GWL_STYLE, Estilo)
  vr = SendMessage(hWnd, EM_SETPASSWORDCHAR, Máscara, 0)
End Sub
```

MOVIENDO CONTROLES ENTRE FORMAS

Visual Basic no soporta el movimiento de controles entre formas. No obstante, creando en cada forma un array de controles del mismo tipo que el control que se quiere mover, se puede simular el movimiento de un control de una forma a otra. Para conseguir ésto, hay que utilizar las siguientes funciones de la API de Windows: **GetFocus** y **GetParent**.

Para simular el movimiento de controles entre formas, realice los pasos siguientes:

1. Inicie una nueva aplicación. Esto hace que se cree una forma *Form1*.
2. Cree una nueva forma *Form2* (orden **New Form** del menú **File**).
3. Cree un nuevo módulo *mover.bas* (orden **New Module** del menú **File**).

```
    Else
      Mover = NO
      Forma.Mover.Caption = "Mover = No"
    End If

    For I = 0 To 4
      Forma.Orden(I).DragMode = Mover
    Next I
End Sub
```

IMPRIMIR UNA IMAGEN VISUAL BASIC

Visual Basic tiene un método denominado **PrintForm** que permite imprimir todos los controles visibles y mapas de bits de una forma. Por ejemplo,

```
Form1.PrintForm
```

El método **PrintForm** también imprime los gráficos e imágenes añadidos/as a la forma en tiempo de ejecución, si la propiedad **AutoRedraw** de los mismos es **True** (–1).

En cambio, no hay forma de imprimir sólo una imagen a menos que se utilicen funciones de la API de Windows. Esto es útil cuando se quiere controlar la posición o el tamaño de la imagen que se desea imprimir; también es útil cuando en una misma hoja se quiere incluir una determinada imagen con otras imágenes o texto.

Para imprimir una imagen, siga los pasos siguientes:

1. Cree un contexto de dispositivo en memoria (memory device context - **mDC**) que sea compatible con el mapa de bits que desea imprimir. Un contexto de dispositivo en memoria es un bloque de memoria que se utiliza para preparar la imagen antes de imprimirla. Para ello, utilice la función de la API de Windows **CreateCompatibleDC**.

   ```
   hmDC = CreateCompatibleDC(Picture1.hDC)
   ```

   Esta función devuelve un **handle** al mDC creado. El **mDC** es compatible con el dispositivo especificado por **hDC**.

2. Seleccione la imagen y muévala al contexto de dispositivo creado. Para realizar esta operación, utilice la función de la API de Windows **SelectObject**.

   ```
   hBitMap = SelectObject(hmDC, Picture1.Picture)
   ```


Si en el **mDC** hay un objeto del mismo tipo, resultado de una selección anterior, éste es removido por el nuevo objeto seleccionado. La función **SelectObject** devuelve el **handle** del objeto removido.

3. Utilice la función **StretchBlt** para copiar el mapa de bits del contexto de dispositivo en memoria, a la impresora. Los argumentos de esta función indican tanto para el destino como para el origen, el dispositivo, esquina superior izquierda y tamaño del rectángulo que contiene la imagen. Esta función devuelve un cero si ocurre un error y un valor distinto de cero en caso contrario.

   ```
   ErrorAPI = StretchBlt(
              Printer.hDC, 0, 0,
              Printer.ScaleWidth, Printer.ScaleHeight,
              hmDC, 0, 0,
              Picture1.ScaleWidth, Picture1.ScaleHeight,
              SRCCOPY)
   ```

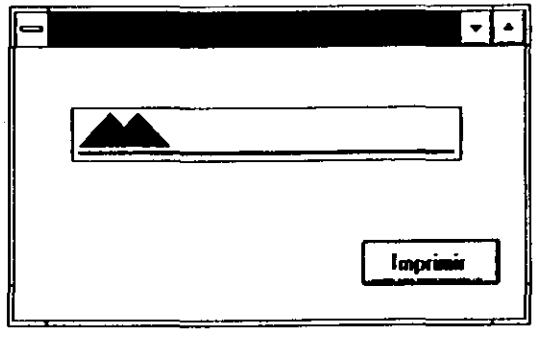
4. Utilice la función **DeleteDC** para borrar el mapa de bits del contexto de dispositivo en memoria. Esta operación exige invocar previamente a la función **SelectObject**, ya que un objeto seleccionado y movido a un contexto de dispositivo no puede borrarse hasta que no sea removido por otro. La función **DeleteDC** devuelve un cero si ocurre un error y un valor distinto de cero en caso contrario.

   ```
   hBitMap = SelectObject(hmDC, hBitMap)
   ErrorAPI = DeleteDC(hmDC)
   ```

Como ejemplo, inicie una nueva aplicación y cree una forma titulada *Form1* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *printer.mak* y la forma con el nombre *printer.frm*.

Objeto	Propiedad	Valor
Imagen	CtlName	Picture1
	AutoRedraw	True
	AutoSize	True
	Picture	...\vb.cbt\brdrlk.bmp
Botón	Caption	Imprimir
	CtlName	Imprimir
	Default	True

Para que este ejemplo trabaje correctamente necesita utilizar una impresora láser estándar (no **PostScript**).



Añada las siguientes declaraciones a la sección de declaraciones de la forma. Recuerde que cada sentencia **Declare** debe estar en una sola línea.

```
Declare Function CreateCompatibleDC Lib "GDI" (
            ByVal hDC As Integer) As Integer

Declare Function SelectObject Lib "GDI" (
            ByVal hDC As Integer,
            ByVal hObject As Integer) As Integer

Declare Function StretchBlt% Lib "GDI" (
            ByVal hDC%, ByVal X%, ByVal Y%,
            ByVal nWidth%, ByVal nHeight%,
            ByVal hSrcDC%, ByVal XSrc%, ByVal YSrc%,
            ByVal nSrcWidth%, ByVal nSrcHeight%,
            ByVal dwRop&)

Declare Function DeleteDC Lib "GDI" (
            ByVal hDC As Integer) As Integer

'Declare Function Escape Lib "GDI" (
            ByVal hDC As Integer,
            ByVal nEscape As Integer,
            ByVal nCount As Integer,
            lpInData As Any,
            lpOutData As Any) As Integer
```

Cuando el usuario ejecute la aplicación y pulse el botón *Imprimir* se obtendrá por la impresora la imagen especificada por el código del procedimiento que se presenta a continuación. La posición y el tamaño de la imagen se especifican en la llamada a la función **StretchBlt**.

Abra la ventana de código correspondiente al botón *Imprimir* y escriba el procedimiento siguiente.

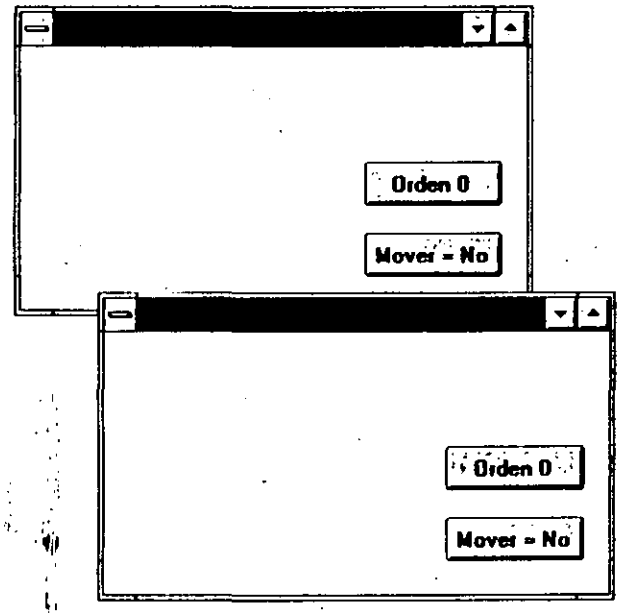

```
Sub Imprimir_Click ()
  Dim hmDC As Integer, hBitMap As Integer, ErrorAPI As Integer
  Const SRCCOPY = &HCC0020 'destino = origen
  Const PIXEL = 3
  Const NEWFRAME = 1 'nueva página

  'Visualizar el reloj de arena
  MousePointer = 11

  'Handle para identificar la imagen
  Picture1.Picture = Picture1.Image

  'La función StretchBlt requiere coordenadas en pixels
  Picture1.ScaleMode = PIXEL
  Printer.ScaleMode = PIXEL

  Printer.Print 'avanzar una línea

  hmDC = CreateCompatibleDC(Picture1.hDC)
  hBitMap = SelectObject(hmDC, Picture1.Picture)

  'Colocar en el destino la imagen centrada y de tamaño 5 veces el real
  ErrorAPI = StretchBlt(Printer.hDC,
    Printer.ScaleWidth \ 2 - Picture1.ScaleWidth * 5 \ 2, 0,
    Picture1.ScaleWidth * 5, Picture1.ScaleHeight * 5,
    hmDC, 0, 0,
    Picture1.ScaleWidth, Picture1.ScaleHeight,
    SRCCOPY)

  hBitMap = SelectObject(hmDC, hBitMap)
  ErrorAPI = DeleteDC(hmDC)

  'Print Escape(Printer.hDC, NEWFRAME, 0, ByVal 0&, ByVal 0&)
  Printer.NewPage
  Printer.EndDoc

  MousePointer = 1 'restaurar el puntero del ratón
End Sub
```

La función **Escape** de la API de Windows permite enviar secuencias de escape a un dispositivo tal como una impresora. Por ejemplo NEWFRAME permite finalizar la página actual y avanzar a la siguiente página. Todas las secuencias de escape soportadas por Windows 3.0, no son soportadas por Windows 3.1. La secuencia NEWFRAME es soportada por Windows 3.0 y 3.1, sin embargo en Windows 3.1 es más usual utilizar las funciones **EndPage** y **StartPage**. En Visual Basic para finalizar la página actual y avanzar a una nueva página tenemos el método **NewPage**.

4. Sobre ambas formas, cree los controles que se indican en la tabla siguiente.

Objeto	Propiedad	Valor
Botón 1	Caption CtlName Index	Orden 0 Orden 0
Botón 2	Caption CtlName	Mover = No Mover

Guarde la aplicación con el nombre *mover.mak* y las formas con los nombres *mover1.frm* y *mover2.frm* respectivamente.



Añada la siguiente declaración al módulo global y denomínelo *mover_g.bas*.

```
Dim I As Integer
```

Cuando el usuario ejecute la aplicación, *Form1* se situará en la mitad izquierda de la pantalla, se visualizará *Form2* y se añadirán al array de controles *Orden()* de *Form1* cuatro botones más titulados *Orden 1*, *Orden 2*, *Orden 3* y *Orden 4*. Para ello, añada a *Form1* el siguiente procedimiento conducido por el suceso **Load**.


```
Sub Form_Load ()
  'Mover la forma a la mitad izquierda de la pantalla
  Move 0, Top
  Form2.Show
  Orden(0).Top = 150
  Orden(0).Left = 150

  For I = 1 To 4 'Cargar el array de controles
    Load Orden(I)
    Orden(I).Top = Orden(I - 1).Top + Orden(I - 1).Height
  Next I

  For I = 1 To 4 'Definir propiedades de los controles
    Orden(I).Caption = "Orden" + Str$(I)
    Orden(I).Visible = -1
  Next I
End Sub
```

Cuando el procedimiento *Form_Load* asociado con la forma *Form1* ejecuta la sentencia *Form2.Show*, se carga la forma *Form2* lo que hace que se ejecute el procedimiento *Form_Load* asociado con ésta. Este procedimiento situará *Form2* en la mitad derecha de la pantalla y añadirá al array de controles *Orden()* de la misma, cuatro botones más titulados *Orden 1*, *Orden 2*, *Orden 3* y *Orden 4*, inicialmente no visibles. Para que se realicen estas operaciones añada a *Form2* el siguiente procedimiento conducido por el suceso **Load**.

```
Sub Form_Load ()
  'Mover la forma a la mitad derecha de la pantalla
  Move Screen.Width \ 2, Top
  Orden(0).Visible = 0

  For I = 1 To 4 'Cargar el array de controles
    Load Orden(I)
    Orden(I).Top = Orden(I - 1).Top + Orden(I - 1).Height
    Orden(I).Visible = 0
  Next I
End Sub
```

Cuando el usuario pulse alguno de los botones *Orden 0* a *Orden 4*, se ejecutará el procedimiento *BotónPulsado* que, simplemente, visualizará un mensaje indicando la orden que se ha ejecutado. Para ello, añada a *Form1* y a *Form2* el siguiente procedimiento conducido por el suceso **Click**. El procedimiento *BotónPulsado* se escribe a continuación en el módulo *mover.bas*.

```
Sub Orden_Click (Index As Integer)
  BotónPulsado Orden(Index)
End Sub
```

Cuando el usuario pulse el botón *Mover = Si/No*, se ejecutará el procedimiento *MoverSiNo* para permitir que los botones *Orden 0* a *Orden 4*, puedan o no moverse de una forma a otra. Para ello, añada a *Form1* el siguiente procedimiento conducido por el suceso **Click**.

```
Sub Mover_Click ()
  Static Mover As Integer
  MoverSiNo Form1, Mover
End Sub
```

Por la misma razón, añada a *Form2* el mismo procedimiento anterior, pero ahora pasando como argumento *Form2*.

```
Sub Mover_Click ()
  Static Mover As Integer
  MoverSiNo Form2, Mover
End Sub
```

La variable estática *Mover* tiene el valor NO (0) cuando el título del botón es *Mover = No* y tiene el valor SI (1) cuando el título del botón es *Mover = Si*.

El procedimiento *MoverSiNo* se escribe a continuación en el módulo *mover.bas*.

Cuando el botón *Mover = Si/No* tenga por título *Mover = Si*, el usuario podrá apuntar a cualquiera de los botones *Orden 0* a *Orden 4* de la forma correspondiente y arrastrarlo para dejarlo sobre la otra. Cuando realice esta operación, se ejecutará el procedimiento conducido por el suceso **DragDrop** para el objeto sobre el que se arrastra, el cual invoca al procedimiento *MoverControl* que elimina el control arrastrado de la forma origen y lo añade a la forma destino. El procedimiento *MoverControl* se escribe a continuación en el módulo *mover.bas*. Para mover un control de *Form2* a *Form1*, añada a *Form1* el siguiente procedimiento.

```
Sub Form_DragDrop (Source As Control, X As Single, Y As Single)
  Dim hWndCtrl As Integer, hWndPadre As Integer
  Source.SetFocus
  hWndCtrl = GetFocus()
  hWndPadre = GetParent(hWndCtrl
  If hWndPadre <> Form1.hWnd Then
    MoverControl Source, Form1
  End If
End Sub
```


Para mover un control de *Form1* a *Form2*, añada a *Form2* el siguiente procedimiento conducido por el suceso **Click**.

```
Sub Form_DragDrop (Source As Control, X As Single, Y As Single)
  Dim hWndCtrl As Integer, hWndPadre As Integer
  Source.SetFocus
  hWndCtrl = GetFocus()
  hWndPadre = GetParent(hWndCtrl)
  If hWndPadre <> Form2.hWnd Then
    MoverControl Source, Form2
  End If
End Sub
```

Para finalizar la aplicación, añada un nuevo módulo llamado *mover.bas* y escriba en él, las siguientes declaraciones, definiciones, funciones y procedimientos referenciadas/os en los procedimientos hasta ahora expuestos.

```
'Declaraciones de las funciones de la API de Windows
Declare Function GetFocus Lib "User" () As Integer
Declare Function GetParent Lib "User" (
                    ByVal hWnd As Integer) As Integer
```

Windows mantiene una tabla de todos los **handles**, así como de los enlaces existentes entre los objetos padres e hijos. La función **GetParent** devuelve el **handle** del padre del objeto referenciado por *hWnd*.

```
Sub BotónPulsado (Ctrl As Control)
  MsgBox "Orden" + Str$(Ctrl.Index) + " pulsada!!"
End Sub

Sub MoverControl (Ctrl As Control, Destino As Form)
  Dim Indice As Integer
  Indice = Ctrl.Index
  Destino.Orden(Indice).Caption = Ctrl.Caption
  Destino.Orden(Indice).Left = Ctrl.Left
  Destino.Orden(Indice).Top = Ctrl.Top
  Destino.Orden(Indice).Width = Ctrl.Width
  Destino.Orden(Indice).Height = Ctrl.Height
  Destino.Orden(Indice).Visible = -1
  Ctrl.Visible = 0
End Sub

Sub MoverSiNo (Forma As Form, Mover As Integer)
  Const NO = 0, SI = 1

  If Mover = NO Then
    Mover = SI
    Forma.Mover.Caption = "Mover = Si"
```

El objeto **Printer** es utilizado para controlar el texto y los gráficos que se imprimen en una página y para enviarlos directamente a la impresora por defecto.

El método **EndDoc** asociado con el objeto **Printer** finaliza un documento enviado a la impresora, descargándolo en dicho dispositivo o en la cola de impresión.

Si se invoca **EndDoc** inmediatamente después del método **NewPage**, entonces no se añade una página en blanco.

CAPÍTULO 15

COMUNICACIONES

INTRODUCCIÓN

Este capítulo presenta técnicas de comunicación con DOS, ejecutando programas DOS desde Visual Basic, y técnicas de comunicación con el exterior utilizando periféricos como el puerto serie, el altavoz, la impresora y el monitor.

EJECUTAR UN PROGRAMA DOS

- Supongamos que tenemos una utilidad DOS que en ocasiones necesitamos ejecutar desde Windows. Para no forzar al usuario a invocar al **shell** del DOS, escribir la orden y ejecutarla, implementaremos una aplicación para que pulsando simplemente un botón, se muestren los resultados requeridos. Por otra parte, una vez que el usuario ejecute la utilidad DOS, cómo se entera la aplicación Visual Basic que dicha utilidad ha finalizado (comunicación entre programas).

- Para ilustrar esto, cree primero un fichero denominado PROGDOS.BAT que contenga la línea que se indica a continuación.

```
DIR | FIND "<DIR>" | SORT > DIR.TXT
```

Esta utilidad DOS crea el fichero DIR.TXT con los nombres de los subdirectorios del directorio actual, ordenados alfabéticamente.

A continuación, utilizando el editor PIF de Windows, cree un fichero denominado PROGDOS.PIF con el contenido siguiente:

Nombre del programa: PROGDOS.BAT

Título de la ventana: directorios
Directorio inicial: C:\VB\SAMPLES\CAP15

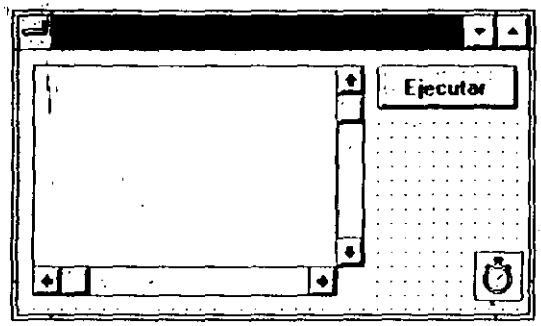
y elija las opciones,

Uso en pantalla: En ventana
Ejecución: Segundo plano

Este fichero es necesario para que puedan ser utilizados los estilos de la sentencia **Shell**. Por ejemplo, el estilo 2 significa ventana minimizada (transformada en un icono) y enfocada.

Inicie una nueva aplicación y cree una forma titulada *Ejecutar un programa DOS* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *dos.mak* y la forma con el nombre *dos.frm*. Posteriormente presentaremos otras dos versiones de ésta misma aplicación con los nombres *dos1.mak* y *dos2.mak*.

Objeto	Propiedad	Valor
Caja de texto	CtlName	Text1
	Multiline	True
	ScrollBars	3 - Both
	Text	(nada)
Botón	Caption	Ejecutar
	CtlName	Ejecutar
Temporizador	CtlName	Timer1
	Enabled	False
	Interval	1000



Escriba en la sección de declaraciones de la forma las siguientes sentencias:


```
Declare Function GetNumTasks Lib "Kernel" () As Integer
Dim NumTareas As Integer, Id As Integer
Const True = -1, False = 0
```

La función **GetNumTasks** de la API de Windows, devuelve como resultado el número de tareas que están actualmente ejecutándose.

Cuando el usuario pulse el botón *Ejecutar*, el procedimiento *Ejecutar_Click* tomará el número de tareas que se están ejecutando, ejecutará la utilidad DOS invocada a través de *progdos.pif*, y activará el temporizador que hace que el procedimiento *Timer1_Timer* se ejecute cada segundo (propiedad **Interval**). Este procedimiento tiene como finalidad verificar cuando finaliza la utilidad DOS, para a continuación visualizar el contenido del fichero DIR.TXT.

```
Sub Ejecutar_Click ()
  Screen.MousePointer = 11 'reloj de arena
  NumTareas = GetNumTasks() 'tareas ejecutándose
  Text1.Text = ""
  ChDir ("c:\vb\samples\cap15")
  Id = Shell("progdos.pif", 2) 'se ejecuta progdos.bat
  Timer1.Enabled = True 'temporizador activo
End Sub
```

Esto es así, porque si nada mas ejecutar la sentencia **Shell** abriéramos el fichero DIR.TXT para visualizarlo, nos encontraríamos con un fichero inexistente lo que daría lugar a un error. Esto ocurre porque una vez que Visual Basic ha lanzado un proceso DOS con **Shell**, pasa a ejecutar la siguiente sentencia sin esperar a que finalice dicho proceso DOS.

Para que el proceso Visual Basic no continúe mientras no finalice el proceso DOS, utilizaremos un temporizador que ejecutará a intervalos de tiempo un procedimiento que comprobará este hecho. Este procedimiento abrirá el fichero DIR.TXT y lo visualizará cuando el número de tareas en ejecución sea el inicial. Escriba las siguientes sentencias para el procedimiento *Timer1_Timer*.

```
Sub Timer1_Timer ()
  Dim Línea As String, CRLF As String
  CRLF = Chr$(13) + Chr$(10)

  If GetNumTasks() <> NumTareas Then
    Exit Sub
  Else
    'Abrir el fichero creado y visualizar su contenido en Text1
    Open "dir.txt" For Input As #1
    Do While Not EOF(1)
      Line Input #1, Línea
```

```
    Text1.SelText = Text1.SelText - Línea - CRLF
  Loop
  Close #1

  Timer1.Enabled = 0
  Screen.MousePointer = 1
 End If
End Sub
```

Otra forma de realizar esta misma aplicación es prescindir del temporizador y utilizar **GetNumTasks** en un lazo para verificar cuando el número de tareas en ejecución es el inicial. En el cuerpo de este lazo colocaremos la función **DoEvents** para permitir que otras aplicaciones compartan los recursos del ordenador. De esta forma el procedimiento *Timer1_Timer* queda eliminado y el procedimiento *Ejecutar_Click* queda como se indica a continuación.

```
Sub Ejecutar_Click ()
  Dim Línea As String, CRLF As String
  CRLF = Chr$(13) + Chr$(10)

  Screen.MousePointer = 11 'reloj de arena
  NumTareas = GetNumTasks() 'tareas ejecutándose
  Text1.Text = ""
  ChDir ("c:\vb\samples\cap15")
  Id = Shell("progdos.pif", 2) 'se ejecuta progdos.bat

  'Esperar a que finalice la tarea DOS
  Do While GetNumTasks() <> NumTareas
    Id = DoEvents()
  Loop

  'Abrir el fichero creado y visualizar su contenido en Text1
  Open "dir.txt" For Input As #1
  Do While Not EOF(1)
    Line Input #1, Línea
    Text1.SelText = Text1.SelText + Línea + CRLF
  Loop
  Close #1
  Screen.MousePointer = 1
End Sub
```

Esta forma de verificar cuando una tarea DOS termina tiene un problema y es que durante la ejecución de la misma comience otra tarea. Para solucionar este problema existe una forma más segura de realizar esta verificación que consiste en la utilización de dos funciones de la API de Windows denominadas **GetActiveWindow** y **IsWindow**.


```
Declare Function GetActiveWindow Lib "User" () As Integer
Declare Function IsWindow Lib "User" (ByVal hWnd As Integer) As Integer
```

La función **GetActiveWindow** devuelve el **handle** correspondiente a la ventana que está actualmente activa (la ventana que tiene el foco) y la función **IsWindow** devuelve un valor **True** (-1) si la ventana especificada existe y un valor **False** (0) si no existe.

Para ilustrar esta nueva forma de proceder, escriba en la sección de declaraciones de la forma las siguientes sentencias:

```
Declare Function GetActiveWindow Lib "User" () As Integer
Declare Function IsWindow Lib "User" (ByVal hWnd As Integer) As Integer
Dim hWndAct As Integer, Id As Integer
```

Cuando el usuario pulse el botón *Ejecutar* tienen que sucederse los siguientes hechos: primero se lanza el proceso DOS con **Shell** para su ejecución; una vez activo este proceso, se toma el **handle** de la ventana activa, que en este instante es la ventana correspondiente a dicho proceso DOS; y por último se establece un lazo con **IsWindow** que verifique cuando la ventana deja de existir, lo que ocurrirá cuando finalice la aplicación DOS. En el cuerpo de este lazo se coloca la función **DoEvents** para permitir que otras aplicaciones compartan los recursos del ordenador.

```
Sub Ejecutar_Click ()
  Dim Línea As String, CRLF As String
  CRLF = Chr$(13) + Chr$(10)

  Screen.MousePointer = 11 'reloj de arena
  Text1.Text = ""
  ChDir ("c:\vb\samples\cap15")

  hWndAct = GetActiveWindow() 'handle de la ventana activa
  Id = Shell("progdos.pif", 2) 'se ejecuta progdos.bat
  'Esperar a que se active la tarea DOS
  Do While GetActiveWindow() = hWndAct
    Id = DoEvents()
  Loop

  hWndAct = GetActiveWindow() 'handle de la ventana DOS
  'Esperar a que finalice la tarea DOS
  Do While IsWindow(hWndAct)
    Id = DoEvents()
  Loop

  'Abrir el fichero creado y visualizar su contenido en Text1
  Open "dir.txt" For Input As #1
```

```
  Do While Not EOF(1)
    Line Input #1, Línea
    Text1.SelText = Text1.SelText + Línea + CRLF
  Loop
  Close #1

  Screen.MousePointer = 1
End Sub
```

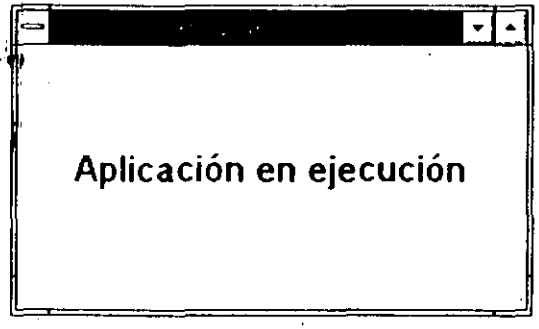
CARGAR UNA APLICACIÓN UNA SOLA VEZ

Seguramente, en alguna ocasión se habrá encontrado con alguna aplicación Windows que permite ejecutarse múltiples veces y también se habrá encontrado con aplicaciones Windows que una vez que están en ejecución, si intenta ejecutarlas de nuevo visualizan un mensaje indicando que dicha aplicación ya está cargada.

Nuestro propósito es mostrar la forma de evitar que una aplicación sea cargada más de una vez.

Inicie una nueva aplicación y cree una forma titulada *Mi Aplicación* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *cargar.mak* y la forma con el nombre *cargar.frm*.

Objeto	Propiedad	Valor
Etiqueta	Caption	Aplicación en ejecución
	AutoSize	True
	FontSize	15



Una forma simple de verificar el número de veces que nuestra aplicación está en ejecución es utilizando las funciones **GetModuleHandle** y **GetModuleUsage**.

Un módulo es una aplicación en ejecución; la función **GetModuleHandle** da el **handle** del módulo especificado y la función **GetModuleUsage** devuelve el número de veces que el módulo especificado ha sido cargado por la misma u otras aplicaciones.

Otro detalle a tener en cuenta es que Visual Basic, por defecto, inicia la ejecución de una aplicación por la primera forma colocada durante el diseño de la aplicación. Para seleccionar otra forma, ejecute la orden **Set Startup Form** del menú **Run**. De esta forma solamente puede cargarse una forma; el resto de las formas deben cargarse explícitamente utilizando la sentencia **Load** o el método **Show**. También puede seleccionarse que Visual Basic inicie la ejecución de la aplicación por un procedimiento previamente creado y denominado **Main** (opción **Sub Main** de la misma orden anterior). En este caso no se carga ninguna forma automáticamente.

Ejecute la orden **Set Startup Form** del menú **Run** y elija la opción **Sub Main** para que la aplicación inicie la ejecución por el módulo **Main** que se presenta a continuación.

Seleccione la orden **New Module** del menú **File** y escriba el código que se muestra a continuación.

```
DefInt A-Z

Declare Function GetModuleHandle Lib "Kernel" (
            ByVal lpModuleName As String) As Integer

Declare Function GetModuleUsage Lib "Kernel" (
            ByVal hModule As Integer) As Integer

Sub Main ()
  hModule = GetModuleHandle("cargar.exe")
  NumCargas = GetModuleUsage(hModule)
  If NumCargas > 1 Then
    MsgBox "Esta aplicación está actualmente en ejecución"
    End
  Else
    Form1.Show
  End If
End Sub
```

Guarde la aplicación *cargar* y ejecute la orden **Make EXE File** del menú **File** para crear un fichero *cargar.exe* ejecutable desde Windows. Asegúrese de que este fichero y el fichero **vbrun100.dll** se encuentran en el mismo directorio. Ahora simplemente diríjase al **Administrador de archivos**, ejecute la aplicación

y reduzca a un icono la forma resultante. A continuación, pruebe a ejecutar de nuevo la misma aplicación y verá el mensaje "Esta aplicación está actualmente en ejecución".

SONIDO CON VISUAL BASIC

En Microsoft Basic, QBasic o QuickBasic disponemos de las sentencias SOUND y PLAY para generar sonidos a través del altavoz. Sin embargo, estas sentencias no están disponibles en Visual Basic. La siguiente aplicación tiene como finalidad desarrollar dos procedimientos **Sound** y **Play** que realicen la misma función que las sentencias Basic del mismo nombre.

En el fichero SYSTEM.INI se especifica el **driver** de sonido que utiliza Windows, que por defecto es SOUND.DRV. Este fichero contiene funciones de bajo nivel para controlar el altavoz del PC y se encuentra en el directorio SYSTEM.

El altavoz del PC puede tocar sólo una nota cada vez. En cambio, existen tarjetas de sonido, por ejemplo **Sound Blaster Card**, que típicamente permiten enviar entre 8 y 16 sonidos cada vez. Cada uno de estos tipos de sonido es conocido en Windows como una voz. Cada voz tiene asociada una cola de notas (lista de notas) para tocar, almacenada en memoria. Cuando todas las notas de una cola han sido tocadas, esta queda vacía.

La función **OpenSound** permite que una aplicación pueda acceder al dispositivo de sonido. Esta función retorna el número de voces disponibles. Para el caso del altavoz de un PC este valor es 1 si el dispositivo de sonido está libre y un valor negativo si el dispositivo ya ha sido abierto por otra aplicación. El acceso al dispositivo de sonido está restringido a una sola aplicación. Para abandonar el dispositivo de sonido y liberar la memoria que ocupan las notas, la aplicación tiene que llamar a la función **CloseSound**.

La función **SetVoiceQueueSize** permite fijar el tamaño de memoria en bytes de la cola, correspondiente a una determinada voz, que va a contener las notas. Cada nota ocupa seis bytes de memoria. Para añadir una nota a la cola hay que utilizar la función **SetVoiceNote**. La función **SetVoiceQueueSize** tiene los siguientes parámetros: voz y el tamaño de la cola en bytes. Esta función devuelve un cero si se ejecuta satisfactoriamente.

La función **SetVoiceAccent** permite fijar el acento de una voz. Tiene los siguientes parámetros: voz, tiempo o número de cuartas notas por minuto, nivel de volumen (0 a 255), un valor que afecta a cómo suena cada nota (normal, legato, staccato) y un valor de 0 a 83 para añadir a la nota. Esta función devuelve un cero si se ejecuta satisfactoriamente.

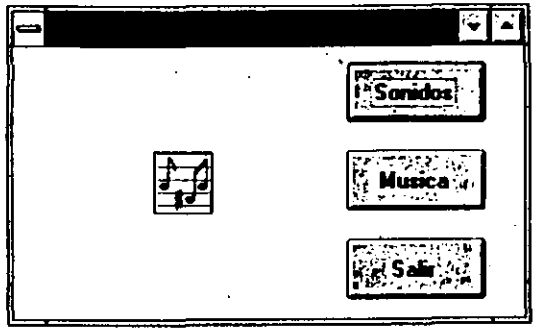
La función **SetVoiceNote** añade una nota a la cola de una determinada voz. Tiene los siguientes parámetros: voz, número de la nota (1 a 84; 0 significa pausa), longitud de la nota (una longitud de 1 equivale a una duración de una nota completa, 2 a media nota, 4 a un cuarto de nota, etc.) y el número de incrementos de media duración que se añaden a la duración de la nota. Esta función devuelve un cero si se ejecuta satisfactoriamente.

La función **SetVoiceSound** permite cambiar la frecuencia utilizada por un dispositivo de sonido para producir un sonido. El cambio se hace para la posición actual de la cola. Tiene los siguientes parámetros: voz, frecuencia en ciclos por segundo y duración del sonido en tics de reloj. La frecuencia es un dato de tipo **long** donde la palabra de mayor peso contiene la frecuencia y la de menor peso la parte fraccionaria que generalmente es cero. Esta función devuelve un cero si se ejecuta satisfactoriamente.

La función **StartSound** comienza a tocar las notas de todas las colas.

Inicie una nueva aplicación y cree una forma titulada *Sonido* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *sonido.mak* y la forma con el nombre *sonido.frm*.

Objeto	Propiedad	Valor
Botón	Caption CtlName	Sonidos Sonidos
Botón	Caption CtlName	Música Música
Botón	Caption CtlName	Salir Salir
Imagen	AutoSize CtlName Picture	True Picture1 ...\vb\icons\mise\mise31



Escriba en la sección de declaraciones de la forma la línea siguiente:

```
DefInt A-Z
```

Para generar un sonido (SOUND) hay que realizar los siguientes pasos:

1. Solicitar acceso al dispositivo de sonido (función **OpenSound**).
2. Mandar un tono de una duración determinada (función **SetVoiceSound**).
3. Indicar a Windows que envíe el sonido al altavoz (función **StartSound**).
4. Repetir los pasos 2 y 3 hasta finalizar.
5. Dejar libre el dispositivo de sonido (función **CloseSound**)

Abra la ventana de código asociada con el botón *Sonidos* y escriba el código que se indica a continuación.

```
Sub Sonidos_Click ()
  'Solicitar acceso al dispositivo de sonido
  Voces = OpenSound()

  For X = 1 To 3
    For J = 4000 To 1000 Step -1
      Sound J, 1 'enviar tono y duración
    Next J
    For J = 140 To 90 Step -1
      Sound J, 18 'enviar tono y duración
    Next J
  Next X
End Sub
```

El procedimiento *Sound*, que se describe un poco más adelante, utiliza las funciones de la API de Windows **SetVoiceSound** para mandar un tono de una duración determinada y **StartSound** para enviar dicho sonido al altavoz.

Cuando finalice la aplicación, pulsando el botón *Salir*, deje libre el dispositivo de sonido para otras aplicaciones.

```
Sub Salir_Click ()
  Unload Form1
End Sub

Sub Form_Unload (Cancel As Integer)
  CloseSound
End Sub
```

Inicie un nuevo módulo (orden **New Module** del menú **File**) y escriba el procedimiento *Sound* que se indica a continuación. Este procedimiento realiza la misma función que la sentencia Basic SOUND.

Para dar independencia al módulo *Sound* realice las declaraciones que se indican a continuación, en la propia sección de declaraciones del módulo.

```
DefInt A-Z

'  Funciones para sonido
'
Declare Function OpenSound Lib "Sound" () As Integer
Declare Function SetVoiceSound Lib "Sound" (
                ByVal nVoice As Integer,
                ByVal lFrequency As Long,
                ByVal nDuration As Integer) As Integer
Declare Function StartSound Lib "Sound" () As Integer
Declare Sub CloseSound Lib "Sound" ()

Sub Sound (ByVal Frecuencia As Long, ByVal Duración As Integer)
  Frecuencia = Frecuencia * 65536
  'Enviar tono y duración
  vr = SetVoiceSound(1, Frecuencia, Duración)
  vr = StartSound() 'tocar el tono enviado
End Sub
```

Para utilizar el módulo *Sound.bas* en otras aplicaciones de la misma forma que lo hemos hecho en ésta, simplemente tiene que añadirlo utilizando la orden **Add File** del menú **File**.

Para generar música (PLAY) hay que realizar los siguientes pasos:

1. Solicitar acceso al dispositivo de sonido (función **OpenSound**).
2. Establecer el tamaño de la cola de notas (función **SetVoiceQueueSize**).

3. Mandar a la cola asociada con una determinada voz, el acento, cuando se indique, y las notas que se quieren tocar (funciones **SetVoiceAccent** y **SetVoiceNote**).
4. Indicar a Windows que envíe la música al altavoz. (función **StartSound**).
5. Dejar libre el dispositivo de sonido (función **CloseSound**) cuando finalice de tocar la música.

Abra la ventana de código asociada con el botón *Música* y escriba el código que se indica a continuación.

```
Sub Música_Click ()
  Parte1$ = "L8GFE-FGGGP8FFF4"
  Parte2$ = "GB-4B-4GFE-FGGGGFFGFE-.P3"
  Melodía$ = "T10003" + Parte1$ + Parte2$
  Play Melodía$
End Sub
```

El procedimiento *Play*, que se describe un poco más adelante, recibe la cadena de caracteres *Melodía* que será analizada carácter a carácter con el fin de enviar las notas y demás información, a la cola correspondiente; la rutina *MandarNota* se encarga de esta operación. Observe que hay veces que es necesario convertir un número dado en la cadena de caracteres a valor numérico; de esto se encarga la rutina *ValorNumérico*. Por ejemplo, cuando el carácter analizado es una nota, el código no envía inmediatamente la nota, sino que espera por el número de incrementos y la longitud de la nota.

Para desarrollar el código correspondiente al análisis de la cadena *Melodía* debe tener un perfecto conocimiento de la sentencia PLAY de Basic y de las notas posibles.

Nota	Referencia	Valor
DO	C	1
	C#	2
RE	D	3
	D#	4
MI	E	5
FA	F	6
	F#	7
SOL	G	8
	G#	9
LA	A	10
	A#	11
SI	B	12

La tabla anterior muestra las doce notas y sus valores asociados correspondientes a la octava cero. Como hay siete octavas, existen 84 notas posibles. Un valor cero se considera una pausa (P). Para calcular el valor correspondiente a una nota se utiliza el código siguiente:

```
Const Notas$ = "PCsDsEFsGsAsB"
```

En la cadena *Notas*, la posición de la nota menos uno coincide con el valor de la nota (*s* indica sostenido, esto es, después de *C* va *C#*). *P* indica pausa.

```
Car$ = Mid$(MelodíaS, PosCar, 1) 'tomar un carácter
Select Case Car$
  Case "A" To "G", "P"
    ...
    Tono = InStr(Notas$, Car$) - 1
    ...
If Tono > 0 Then Tono = (Tono + (Octava * 12)) - 1
```

En conclusión, antes de enviar una nota o el acento, hay que calcular los valores de los parámetros de la función de la API de Windows implicada.

Cuando finalice la aplicación, pulsando el botón *Salir*, deje libre el dispositivo de sonido para otras aplicaciones. Los procedimientos correspondientes a este botón ya fueron descritos anteriormente.

Inicie un nuevo módulo (orden **New Module** del menú **File**) y escriba el procedimiento *Play* que se indica a continuación. Este procedimiento realiza la misma función que la sentencia Basic PLAY.

Para dar independencia al módulo *Play* realice las declaraciones que se indican a continuación, en la propia sección de declaraciones del módulo.

```
DefInt A-Z
'
'  Funciones para sonido
'
Declare Function OpenSound Lib "Sound" () As Integer
Declare Function SetVoiceQueueSize Lib "Sound" (
                ByVal nVoice As Integer,
                ByVal nBytes As Integer) As Integer
Declare Function SetVoiceAccent Lib "Sound" (
                ByVal nVoice As Integer,
                ByVal nTempo As Integer,
                ByVal nVolume As Integer,
                ByVal nMode As Integer,
                ByVal nPitch As Integer) As
```

```
Declare Function SetVoiceNote Lib "Sound" (
                ByVal nVoice As Integer,
                ByVal nValue As Integer,
                ByVal nLength As Integer,
                ByVal nDots As Integer) As Integer
Declare Function StartSound Lib "Sound" () As Integer
Declare Sub CloseSound Lib "Sound" ()
'
'En la cadena Notas, la posición de la nota menos uno
'coincide con el valor de la nota. P = pausa.
'(s indica sostenido, esto es, después de C va C=)
Const Notas$ = "PCsDsEFsGsAsB"
Const Números$ = "0123456789"
'
Const TRUE = -1
Const FALSE = 0

Sub Play (MelodíaS)
  MelodíaS = UCaseS(MelodíaS)
  Voces = OpenSound()
  vr = SetVoiceQueueSize(1, 8192) 'tamaño 8K
  TotalCars = Len(MelodíaS)
  'Valores por defecto
  Tiempo = 120
  Música = 0 ' normal
  Octava = 4
  LongPorDefecto = 4 'duración = 1/4 de nota
  NotaNueva = FALSE
  AcentoNuevo = TRUE
  PosCar = 1
  Do
    CarS = Mid$(MelodíaS, PosCar, 1)
    Select Case CarS
      Case "A" To "G", "P"          'nota o pausa
        GoSub MandarNota
        Tono = InStr(NotasS, CarS) - 1
        NotaNueva = TRUE

      Case "+", "="                 'sostenido
        Tono = Tono + 1

      Case "-"
        Tono = Tono - 1             'bemol

      Case "." 'incrementa en media duración de la nota
        16thMDH = 16thMDH - 1

      Case "1" To "9"               'longitud de la nota
        GoSub ValorNumérico
        LongNotaDada = ValorDeN
```



```
      Case ">"                      'siguiente octava
        GoSub MandarNota
        Octava = Octava + 1

      Case "<"                      'octava anterior
        GoSub MandarNota
        Octava = Octava - 1

      Case "M"                      'música
        GoSub MandarNota
        PosCar = PosCar + 1
        CarS = Mid$(MelodíaS, PosCar, 1)
        Select Case CarS
          Case "N"
            Música = 0 'normal
          Case "L"
            Música = 1 'ligada (legató)
          Case "S"
            Música = 2 'destacada (staccato)
        End Select
        AcentoNuevo = TRUE

      Case "N"                      'nota
        GoSub MandarNota
        PosCar = PosCar + 1
        GoSub ValorNumérico
        Tono = ValorDeN
        NotaNueva = TRUE
        GoSub MandarNota

      Case "O"                      'octava
        GoSub MandarNota
        PosCar = PosCar + 1
        GoSub ValorNumérico
        Octava = ValorDeN

      Case "T"                      'tiempo
        GoSub MandarNota
        PosCar = PosCar + 1
        GoSub ValorNumérico
        Tiempo = ValorDeN
        AcentoNuevo = TRUE

      Case "L"                      'fijar longitud nota
        GoSub MandarNota
        PosCar = PosCar + 1
        GoSub ValorNumérico
        LongPorDefecto = ValorDeN
```

```
        Case Else
      End Select
      PosCar = PosCar + 1
    Loop Until PosCar > TotalCars
    GoSub MandarNota
    vr = StartSound()
  Exit Sub

  MandarNota:
    If AcentoNuevo Then vr = SetVoiceAccent(1, Tiempo, 1, Música, 0)
    If NotaNueva Then
      'Calcular el valor de la nota
      If Tono > 0 Then Tono = (Tono + (Octava * 12)) - 1
      If LongNotaDada Then
        LongitudNota = LongNotaDada
        LongNotaDada = 0
      Else
        LongitudNota = LongPorDefecto
      End If
      vr = SetVoiceNote(1, Tono, LongitudNota, IdeMDN)

      IdeMDN = 0
    End If
    NotaNueva = FALSE
    AcentoNuevo = FALSE
  Return

  ValorNumérico: 'cálculo de n de On, Ln, Tn, etc.
    'Primer dígito (dígito más significativo)
    ValorDeN = Val(Mid$(MelodíaS, PosCar, 1))
    If PosCar < TotalCars Then
      SiguienteCarS = Mid$(MelodíaS, PosCar + 1, 1)
      If InStr(NúmerosS, SiguienteCarS) > 0 Then
        'El segundo carácter es un dígito
        ValorDeN = ValorDeN * 10 + Val(SiguienteCar$)
        PosCar = PosCar + 1
      Else
        Return
      End If
    End If
    If PosCar < TotalCars Then
      SiguienteCarS = Mid$(MelodíaS, PosCar + 1, 1)
      If InStr(NúmerosS, SiguienteCarS) > 0 Then
        'El tercer carácter es un dígito
        ValorDeN = ValorDeN * 10 + Val(SiguienteCar$)
        PosCar = PosCar + 1
      End If
    End If
    Return
  End Sub
```


El **driver** de sonido de Windows 3 hace que la música se procese en segundo plano (**background**), aún cuando la tarea es modal (un tipo de forma que retiene el foco hasta que sea cerrada), por lo que MF y MB son simplemente ignoradas. Por ejemplo, podemos lanzar una música y a continuación visualizar una ventana con un mensaje; la música seguirá sonando hasta finalizar o hasta que se ejecute **CloseSound**.

```
...
StartSound
vr = MsgBox(Mensaje, 48, Título)
CloseSound
```

Este ejemplo, lanza una música y visualiza un mensaje. La música seguirá sonando hasta finalizar o hasta que se cierre la ventana momento en el que se ejecuta **CloseSound**.

Si desea detener la aplicación hasta que la música finalice (música en primer plano) declare la función **CountVoiceNotes**.

```
Declare Function CountVoiceNotes Lib "Sound" (
                 ByVal nVoice As Integer) As Integer
```

y escriba el siguiente código después de cada llamada a *Play*.

```
While CountVoiceNotes(1) > 0: Wend
```

Si además quiere permitir que se ejecuten otras tareas, añada a la condición la función **DoEvents**. Por ejemplo, queremos que al pulsar un botón se visualice un mensaje y la música siga tocando.

```
While DoEvents() And CountVoiceNotes(1) > 0: Wend
```

COMUNICACIONES CON EL PUERTO SERIE

La gestión de los puertos de comunicación no es una tarea fácil. Lo primero que hay que pensar es que los datos llegan a los puertos de forma asíncrona; es decir, su llegada es imprevisible. Esto sugiere que el dato que llega tiene que procesarse inmediatamente puesto que pueden llegar otros datos. De esta tarea se encarga el **hardware** del PC, de forma que cuando detecta la llegada de un dato, interrumpe el flujo normal del proceso para ceder el control a la rutina de proceso de comunicaciones. Esta rutina tiene que ser una rutina de Windows, en lugar de una rutina de la aplicación. Esto es así por dos razones:

- Windows debe mantener el control de la multitarea. En efecto, si la llegada de un dato hiciera que se transfiriera el control del procesador a su aplicación, Windows perdería su habilidad para gestionar la multitarea. Esto quiere decir que Windows tiene que estar entre la aplicación y el **hardware**.

- Windows no puede dirigir el dato recibido directamente a la aplicación. La razón es que los datos que se reciben en el puerto de comunicaciones no llegan con la identidad de la aplicación que los tiene que recibir. Por lo tanto, Windows tiene que salvar en un **buffer** los datos que llegan para una aplicación. Para qué aplicación, debe determinarse por adelantado; esto es, su aplicación debe haberle pedido a Windows la propiedad del puerto, utilizando la función **OpenComm**. Esta misma función establece dos **buffers**, uno para la entrada y otro para la salida.

Cuando una aplicación solicita a Windows la propiedad de un puerto, Windows sólo se lo dará si ninguna otra aplicación lo tiene. Por el mismo motivo, mientras su aplicación tiene el control de un puerto, Windows se lo prohibe a las demás aplicaciones que lo soliciten. Cuando su aplicación finalice la operación de E/S con el puerto, debe dejar el control del mismo para que otras aplicaciones puedan utilizarlo, lo cual requiere llamar a la función **CloseComm**.

Como ejemplo vamos a realizar una aplicación que permita transferir datos entre dos ordenadores personales. Para probar la aplicación, debe conectar vía puerto de comunicaciones los dos ordenadores. Una vez realizada la conexión, asegúrese de que está bien hecha utilizando un paquete de comunicaciones comercial como el programa *Terminal* de Windows.

Una forma de realizar esta aplicación sería utilizar las sentencias estándar de E/S **Open**, **Close**, **Get**, **Put** y **Err**, para efectuar las operaciones de E/S sobre el puerto. Pero esta forma de trabajo presenta problemas de pérdida de datos, especialmente por encima de 2400 baudios; también, cuando Windows cede el control a otra aplicación, todos los caracteres recibidos en esta situación se perderían. La razón es que estas sentencias trabajan con los **drivers** de los puertos serie de DOS y éstos no fueron diseñados ni para velocidades altas, ni para un entorno multitarea como Windows. Por lo tanto, la solución es utilizar las funciones de la API de Windows equivalentes a estas sentencias, las cuales se indican en la tabla siguiente.

VB	API	Función
Open	OpenComm	Abre un puerto de comunicaciones
Close	CloseComm	Cierra un puerto de comunicaciones
Get	ReadComm	Lee datos
Put	WriteComm	Envía datos
Err	GetCommError	Comprueba si ocurrió algún error
	SetCommState	Establece las características del puerto
	BuildCommDCB	Construye el bloque de control de datos

Para establecer una comunicación siga los siguientes pasos:

1. Abra el puerto de comunicación. Para realizar esta operación llame a la función **OpenComm** con los argumentos, puerto de comunicaciones, tamaño del **buffer** de entrada y tamaño del **buffer** de salida. Esta función devuelve un número que identifica al puerto de comunicaciones abierto o un número negativo si ocurre un error (constantes IE_).

2. Construya el bloque de control de datos (DCB). Para ello, llame a la función **BuildCommDCB** con los argumentos, definición del puerto y estructura DCB. La definición del puerto es una cadena de caracteres con un formato igual al utilizado por la orden MODE de DOS, que se utiliza para llenar la estructura de tipo DCB. Por ejemplo "com2:2400,n,8,1".

3. Establezca las características del puerto. Para realizar esta operación llame a la función **SetCommState** pasando como argumento la estructura de datos de tipo DCB, en la que previamente se almacenaron tales características.

4. Cuando quiera enviar datos al puerto de comunicaciones, utilice la función **WriteComm** que tiene los siguientes parámetros: un valor ID que identifica al dispositivo de comunicaciones (este valor es devuelto por la función **OpenComm**), una cadena de caracteres que contiene los caracteres enviados y el número de caracteres enviados. Un valor negativo indica que ha ocurrido un error. El valor absoluto del valor devuelto por la función coincide con el número de caracteres enviados.

5. Establezca un lazo que esté a la espera de los datos que nuestra aplicación espera recibir. Para ello, utilice la función **ReadComm** que tiene los siguientes parámetros: un valor ID que identifica al dispositivo de comunicaciones, una cadena de caracteres que contiene los caracteres recibidos y el número de caracteres recibidos. Un valor negativo indica que ha ocurrido un error. El valor absoluto del valor devuelto por la función coincide con el número de caracteres leídos.

6. Cuando ocurre un error durante una operación de comunicaciones, Windows bloquea el puerto correspondiente, el cual permanecerá bloqueado hasta que se llame a la función **GetCommError**. El error es copiado en una estructura

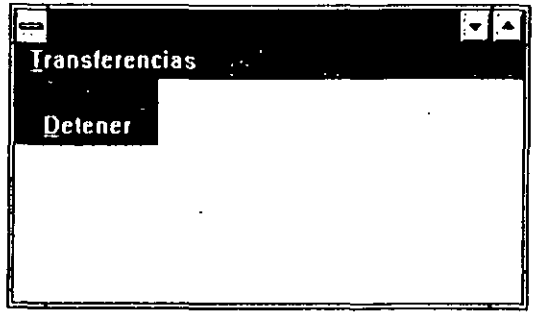
de tipo COMSTAT. Esta función devuelve el código del error ocurrido (constantes CE_).

Es una buena práctica llamar a la función **GetCommError** después de invocar a la función **ReadComm** por si ocurre algún error sobre el dispositivo de comunicaciones.

7. Utilice la función **CloseComm** para cerrar el puerto de comunicaciones cuando éstas finalicen. Si el puerto no puede cerrarse, esta función devuelve un valor negativo.

Inicie una nueva aplicación y cree una forma titulada *Comunicaciones* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *rs232.mak* y la forma con el nombre *rs232.frm*.

Objeto	Propiedad	Valor
Forma	Caption AutoRedraw FormName	Comunicaciones False FormCom
Menú	Caption CtlName	&Transferencias Transferencias
Transferencias: orden 0	Caption CtlName	Tx/Rx &TransRec
Transferencias: orden 1	Caption CtlName	&Detener Detener
Menú	Caption CtlName	Salir &Salir



Escriba las siguientes declaraciones en el módulo global, y denomínelo *rs232.bas*. Estas declaraciones, que han sido importadas del fichero WIN-

DOWS.H, adaptadas al lenguaje de Visual Basic e incluídas en WINAPI.TXT, están relacionadas con las sentencias **Declare** correspondientes a las funciones de comunicaciones de la API de Windows, que se encuentran en el fichero WINAPI.TXT, y que también tiene que añadir a este mismo módulo.

```
' Error Flags

Global Const CE_RXOVER = &H1      ' Receive Queue overflow
Global Const CE_OVERRUN = &H2     ' Receive Overrun Error
Global Const CE_RXPARITY = &H4    ' Receive Parity Error
Global Const CE_FRAME = &H8       ' Receive Framing error
Global Const CE_BREAK = &H10      ' Break Detected
Global Const CE_CTSTO = &H20      ' CTS Timeout
Global Const CE_DSRTO = &H40      ' DSR Timeout
Global Const CE_RLSDTO = &H80     ' RLSD Timeout
Global Const CE_TXFULL = &H100    ' TX Queue is full
Global Const CE_PTO = &H200       ' LPTx Timeout
Global Const CE_IOE = &H400       ' LPTx I/O Error
Global Const CE_DNS = &H800       ' LPTx Device not selected
Global Const CE_OOP = &H1000      ' LPTx Out-Of-Paper
Global Const CE_MODE = &H8000     ' Requested mode unsupported

Global Const IE_BADID = (-1)      ' Invalid or unsupported id
Global Const IE_OPEN = (-2)       ' Device Already Open
Global Const IE_NOPEN = (-3)      ' Device Not Open
Global Const IE_MEMORY = (-4)     ' Unable to allocate queues
Global Const IE_DEFAULT = (-5)    ' Error in default parameters
Global Const IE_HARDWARE = (-10)        ' Hardware Not Present
Global Const IE_BYTESIZE = (-11)        ' Illegal Byte Size
Global Const IE_BAUDRATE = (-12)        ' Unsupported BaudRate

Type DCB
  Id As String * 1          'Internal Device ID
  BaudRate As Integer       'Baudrate at which runing
  ByteSize As String * 1    'Number of bits/byte, 4-8
  Parity As String * 1      '0-4=None,Odd,Even,Mark,Space
  StopBits As String * 1    '0,1,2=1,1.5,2
  RlsTimeout As Integer     'Timeout for RLSD to be set
  CtsTimeout As Integer     'Timeout for CTS to be set
  DsrTimeout As Integer     'Timeout for DSR to be set
  Byte1 As String * 1
  Byte2 As String * 1
  XonChar As String * 1     'Tx and Rx X-ON character
  XoffChar As String * 1    'Tx and Rx X-OFF character
  XonLim As Integer         'Transmit X-ON threshold
  XoffLim As Integer        'Transmit X-OFF threshold
  PeChar As String * 1      'Parity error replacement char
  EofChar As String * 1     'End of Input character
  EvtChar As String * 1     'Recieved Event character
```

```
  TxDelay As Integer      'Amount of time between chars
End Type

Type COMSTAT
  Byte As String * 1
  cbInQue As Integer      'count of characters in Rx Queue
  cbOutQue As Integer     'count of characters in Tx Queue
End Type

'   Funciones para comunicaciones
'
Declare Function OpenComm Lib "User" (
              ByVal lpComName As String,
              ByVal wInQueue As Integer,
              ByVal wOutQueue As Integer) As Integer

Declare Function BuildCommDCB Lib "User" (
              ByVal lpDef As String, lpDCB As DCB) As Integer

Declare Function GetCommState Lib "User" (
              lpDCB As DCB) As Integer

Declare Function ReadComm Lib "User" (
              ByVal nCid As Integer,
              ByVal lpBuf As String,
              ByVal nSize As Integer) As Integer

Declare Function WriteComm Lib "User" (
              ByVal nCid As Integer,
              ByVal lpBuf As String,
              ByVal nSize As Integer) As Integer

Declare Function GetCommError Lib "User" (
              ByVal nCid As Integer,
              lpStat As COMSTAT) As Integer

Declare Function Closecomm Lib "User" (
              ByVal nCid As Integer) As Integer

'   Funciones para desplazamiento sobre una ventana
'
Declare Sub ScrollWindow Lib "User" (
              ByVal hWnd As Integer,
              ByVal XAmount As Integer,
              ByVal YAmount As Integer,
              lpRect As Any,
              lpClipRect As Any)

Decla    Sub UpdateWindow Lib "User" (ByVal hWnd As Integer)
```


Escriba las siguientes declaraciones en la sección de declaraciones de la forma *FormCom*.

```
Dim LpDCB As DCB 'bloque de control de datos
Dim nCid As Integer 'identificación del puerto
Const DefCom = "COM2:9600,n,8,1" 'características del puerto
Const Tb = 1024 'tamaño del buffer
```

Cuando el usuario haga clic en la orden *Tx/Rx* del menú *Transmisiones*, se abre el puerto de comunicaciones COM2 y se establecen sus características; para ello se invoca al procedimiento *Abrir_Com* y se le pasa como argumento la cadena *DefCom* que contiene el nombre del puerto y los parámetros bajo los que se desea establecer la comunicación. En nuestro ejemplo se ha elegido el puerto COM2 (el puerto COM1 generalmente es utilizado para el ratón), velocidad 9600 baudios, no paridad, 8 bits por dato, y 1 bit de parada. A continuación, se llama al procedimiento *RecibirCom* para establecer un lazo que mantenga a la aplicación a la espera de recibir datos.

```
Sub TransRec_Click ()
  Abrir_Com DefCom$
  If nCid Then
    FormCom.Caption = "Puerto de comunicaciones abierto"
  End If
  RecibirCom
End Sub

Sub Abrir_Com (DefPuertoCom$)
  Dim vr As Integer

  'Cerrar cualquier puerto abierto previamente
  vr = Closecomm(1)
  vr = Closecomm(2)

  'Abrir el puerto de comunicaciones. nCid es el ID
  'nCid < 0 indica un error (constantes IE_).

  PuertoCom$ = Left$(DefPuertoCom$, InStr(DefPuertoCom$, ":") - 1)
  nCid = OpenComm(PuertoCom$, Tb, Tb)
  If nCid < 0 Then
    MsgBox "No se puede abrir el puerto COM: " + Str$(nCid), 16
    End
  End If

  'Cargar el bloque de control de datos (DCB) con los
  'valores dados en DefPuertoCom$
  'El primer parámetro del DCB es puesto mar    nente

  LpDCB.Id = Chr$(nCid)
```

```
    If (BuildCommDCB(DefPuertoCom$, LpDCB)) Then
      MsgBox "No se puede construir el DCB", 16
      End
    End If

    'Establecer las características del puerto
    If (SetCommState(LpDCB)) Then
      MsgBox "No se pueden poner los parámetros para COM", 16
      End
    End If
End Sub

Sub RecibirCom ()
  Dim NumCars As Integer, RError As Integer, vr As Integer
  Dim BufferEnt As String * Tb, Lpstat As COMSTAT
  Do While nCid
    'Leer datos del puerto
    NumCars = ReadComm(nCid, BufferEnt, Len(BufferEnt))
    'Ignorar errores
    If NumCars < 0 Then NumCars = -NumCars
    'Visualizar datos
    If NumCars Then VisualizarDatos Left$(BufferEnt, NumCars)
    'Restablecer la comunicación si ocurre un error
    RError = GetCommError(nCid, Lpstat)
    'Permitir que otras aplicaciones se ejecuten
    vr = DoEvents()
  Loop
End Sub
```

Si la variable *NumCars* es negativa es que ha ocurrido un error, no obstante, su valor absoluto indica el número de caracteres leídos. Observe que en caso de error se ignora el signo, porque a continuación siempre se llama a la función **GetCommError**. Esta función devuelve un entero correspondiente al código de error ocurrido (constantes CE_).

Los datos leídos son visualizados en la forma por el procedimiento *VisualizarDatos*, que también provee un sistema de desplazamiento vertical de la información en la ventana, cuando ésta se llena. Para ello, utiliza las funciones de la API de Windows **ScrollWindow** y **UpdateWindow**.

La función **ScroollWindows** permite realizar en una ventana (parámetro *hWnd*) un desplazamiento en la dirección X igual al valor *Xamount* y un desplazamiento en la dirección Y igual al valor *Yamount* (ver sentencia **Declare**). Recuerde que la unidad más pequeña de resolución de un monitor es el **pixel**. Esta función no trabaja bien si la propiedad **AutoRedraw** de la fórma es **True**.

La función **UpdateWindow** actualiza la ventana.


```
Sub VisualizarDatos (Buffer As String)
  Dim Yamount As Integer, N As Integer, vr As Integer
  Scalemode = 3 'escala en pixels
  Yamount = TextHeight("a") 'altura del carácter
  'Procesar todos los caracteres
  For N = 1 To Len(Buffer)
    CS = Mid$(Buffer, N, 1)
    '¿Hay que realizar scroll?
    If CS = Chr$(10) And CurrentY >= ScaleHeight - 2 * Yamount Then
      ScrollWindow FormCom.hWnd, 0, -Yamount, ByVal 0&, ByVal 0&
      UpdateWindow FormCom.hWnd
      vr = DoEvents()
      CurrentX = 0
    ElseIf CS = Chr$(13) Then 'si CR, principio de línea
      CurrentX = 0
    Else
      FormCom.Print CS;
    End If
  Next N
End Sub
```

Si ahora el usuario quiere transmitir información, simplemente tiene que escribirla en la ventana. El procedimiento *Form_KeyPress* conducido por el suceso **KeyPress** se encarga de esta actividad. Para enviar datos a la cola de salida, primero hay que verificar que no está llena, para lo cual se llama a la función **GetCommError** y se comprueba el valor del miembro *cbOutQue* de la estructura *Lpstat*. Si hay espacio, la función **WriteComm** envía los datos al **buffer** de salida y Windows se encarga del resto.

```
Sub Form_KeyPress (KeyAscii As Integer)
  Dim NumCars As Integer, vr As Integer, RError As Integer
  Dim Lpstat As COMSTAT 'estado del puesto
  Static BufferSal$

  If nCid = 0 Then Exit Sub

  BufferSal$ = BufferSal$ + Chr$(KeyAscii)
  vr = GetCommError(nCid, Lpstat)  'estado actual
  '¿queda espacio en el buffer?
  If Lpstat.cbOutQue < Tb Then
    'Enviar caracteres al buffer de salida
    NumCars = WriteComm(nCid, BufferSal$, Len(BufferSal$))
    If NumCars <= 0 Then NumCars = -NumCars 'Ignorar errores
    RError = GetCommError(nCid, Lpstat)
    'Eliminar los caracteres transmitidos
    BufferSal$ = Mid$(BufferSal$, NumCars + 1)
  End If
End Sub
```

Cuando el usuario haga clic en la orden *Detener* del menú *Transmisiones*, se ejecuta el procedimiento *Detener_Click* que invoca al procedimiento *Cerrar_Com* que cierra el puerto y detiene las comunicaciones.

```
Sub Detener_Click ()
  Cerrar_Com
  FormCom.Caption = "Puerto de comunicaciones cerrado"
End Sub

Sub Cerrar_Com ()
  Dim vr As Integer
  if nCid vr = Closecomm(nCid)
  If vr < 0 Then
    MsgBox "No se puede cerrar el puerto COM: " + Str$(vr), 16
    End
  Else
    nCid = 0 'Poner a cero nCid para detener RecibirCom
  End If
End Sub
```

Cuando el usuario quiera finalizar la aplicación hará clic en el menú *Salir*. Como se ve en el procedimiento siguiente, para salir de la aplicación no es necesario ejecutar previamente la orden *Detener*.

```
Sub Salir_Click ()
  Cerrar_Com
  End
End Sub
```

Observará que no se han utilizado las constantes de error. Esto es así, porque la idea era crear una aplicación simple, que funcione, y que al mismo tiempo le muestre los elementos con los que puede trabajar.

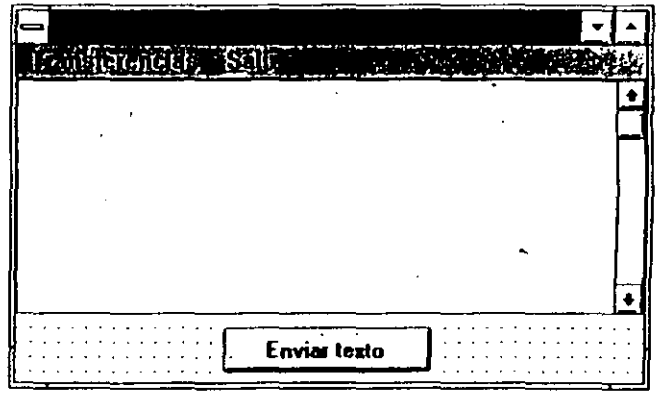
CONTROL DE ERRORES EN LAS CO[illegible]ACIONES

La aplicación siguiente es una nueva versión de la anterior. La idea fundamental [illegible] el tratamiento de los errores durante las comunicaciones.

Ahora, cuando el usuario desee enviar información, la escribirá en una caja de texto multilínea y una vez escrita, pulsará un botón para enviarla. La información recibida se visualizá también sobre la caja de texto.

Inicie una nueva aplicación y cree una forma titulada *Comunicaciones* con los [illegible] que se indican en la tabla siguiente. Guarde la aplicación con el nombre [illegible]*ak* y la forma con el nombre *rs232err.frm*.

Objeto	Propiedad	Valor
Forma	Caption AutoRedraw FormName	Comunicaciones False FormCom
Menú	Caption CtlName	&Transferencias Transferencias
Transferencias: orden 0	Caption CtlName	Tx/Rx &TransRec
Transferencias: orden 1	Caption CtlName	&Detener Detener
Menú	Caption CtlName	Salir &Salir
Caja de texto	CtlName MultiLine ScrollBars Text	[illegible]tl [illegible]ue 2 - Vertical (nada)
Botón	Caption CtlName	Enviar texto TransmitirCom



Escriba las siguientes declaraciones en el [illegible]global, y denomínelo *rs232err.bas*. Estas declaraciones, han sido importadas del fichero WINAPI.TXT.

```
' Constantes Boolean
Global Const FALSE = 0
Global Const TRUE = Not FALSE

' Errores en la comunicación
Global Const CE_RXOVER = &H1    ' Receive Queue o[illegible]low
```

```
Global Const CE_OVERRUN = &H2     '  Receive Overrun Error
Global Const CE_RXPARITY = &H4    '  Receive Parity Error
Global Const CE_FRAME = &H8       '  Receive Framing error
Global Const CE_BREAK = &H10      '  Break Detected
Global Const CE_CTSTO = &H20      '  CTS Timeout
Global Const CE_DSRTO = &H40      '  DSR Timeout
Global Const CE_RLSDTO = &H80     '  RLSD Timeout
Global Const CE_TXFULL = &H100    '  TX Queue is full
Global Const CE_PTO = &H200       '  LPTx Timeout
Global Const CE_IOE = &H400       '  LPTx I/O Error
Global Const CE_DNS = &H800       '  LPTx Device not selected
Global Const CE_OOP = &H1000      '  LPTx Out-Of-Paper
Global Const CE_MODE = &H8000     '  Requested mode unsupported

'  Errores en OpenComm
Global Const IE_BADID = (-1)      '  Invalid or unsupported id
Global Const IE_OPEN = (-2)       '  Device Already Open
Global Const IE_NOPEN = (-3)      '  Device Not Open
Global Const IE_MEMORY = (-4)     '  Unable to allocate queues
Global Const IE_DEFAULT = (-5)    '  Error in default parameters
Global Const IE_HARDWARE = (-10)        '  Hardware Not Present
Global Const IE_BYTESIZE = (-11)        '  Illegal Byte Size
Global Const IE_BAUDRATE = (-12)        '  Unsupported BaudRate

'  Declaraciones para COM
'  Parity: 0-4=None,Odd,Even,Mark,Space
Global Const NOPARITY = 0
Global Const ODDPARITY = 1
Global Const EVENPARITY = 2
Global Const MARKPARITY = 3
Global Const SPACEPARITY = 4

'  Stop Bits: 0,1,2=1,1.5,2
Global Const ONESTOPBIT = 0
Global Const ONE5STOPBITS = 1
Global Const TWOSTOPBITS = 2

'  Estructura de datos DCB (Bloque de Control del Dispositivo)
Type DCB
  Id As String * 1          'Internal Device ID
  BaudRate As Integer       'Baudrate at which runing
  ByteSize As String * 1    'Number of bits/byte, 4-8
  Parity As String * 1      '0-4=None,Odd,Even,Mark,Space
  StopBits As String * 1    '0,1,2=1,1.5,2
  RlsTimeout As Integer     'Timeout for RLSD to be set
  CtsTimeout As Integer     'Timeout for CTS to be set
  DsrTimeout As Integer     'Timeout for DSR to be set
  ModeControl As Integer    'Mode Control Bits Fields
  XonChar As String * 1     'Tx and Rx X-ON character
  XoffChar As String * 1    'Tx and Rx X-OFF character
```



```
  XonLim As Integer         'Transmit X-ON threshold
  XoffLim As Integer        'Transmit X-OFF threshold
  PeChar As String * 1      'Parity error replacement char
  EofChar As String * 1     'End of Input character
  EvtChar As String * 1     'Recieved Event character
  TxDelay As Integer        'Amount of time between chars
End Type

Type COMSTAT
  ModeControl As String * 1
  cbInQue As Integer        'count of characters in Rx Queue
  cbOutQue As Integer       'count of characters in Tx Queue
End Type

'   Funciones para comunicaciones
'
Declare Function OpenComm Lib "User" (
              ByVal lpComName As String,
              ByVal wInQueue As Integer,
              ByVal wOutQueue As Integer) As Integer
Declare Function BuildCommDCB Lib "User" (
              ByVal lpDef As String,
              lpDCB As DCB) As Integer
Declare Function SetCommState Lib "User" (
              lpDCB As DCB) As Integer
Declare Function ReadComm Lib "User" (
              ByVal nCid As Integer,
              ByVal lpBuf As String,
              ByVal nSize As Integer) As Integer
Declare Function WriteComm Lib "User" (
              ByVal nCid As Integer,
              ByVal lpBuf As String,
              ByVal nSize As Integer) As Integer
Declare Function GetCommError Lib "User" (
              ByVal nCid As Integer,
              lpStat As COMSTAT) As Integer
Declare Function FlushComm Lib "User" (
              ByVal nCid As Integer,
              ByVal nQueue As Integer) As Integer
Declare Function Closecomm Lib "User" (
              ByVal nCid As Integer) As Integer
```

Escriba las siguientes declaraciones en la sección de declaraciones de la forma *FormCom*.

```
Dim LpDCB As DCB 'Bloque de control de datos
Dim nCid As Integer 'identificación del puerto
Dim BufferSal As String 'buffer de salida
Const DefCom = "COM2:9600,N,8,1" 'características del puerto
```

```
Const Tb = 2048 'tamaño del buffer de E y de S
```

La inicialización del puerto se hace llamando a la función de la API de Windows **SetCommState** y pasando como argumento la estructura *LpDCB*. La definición del puerto es la cadena *DefCom* que se pasa como argumento cuando se invoca la función de la API de Windows **BuildCommDCB**.

Los caracteres que el usuario escribe sobre *Text1* para transmitir, son almacenados en la variable *BufferSal*; esta variable es estática por defecto. Para ello, escriba el siguiente procedimiento:

```
Sub Text1_KeyPress (KeyAscii As Integer)
  'Caracteres para transmitir
  If KeyAscii = 13 Then ' CR
    BufferSal = BufferSal + Chr$(13) + Chr$(10) ' -CR+LF
  Else
    BufferSal = BufferSal + Chr$(KeyAscii)
  End If
End Sub
```

Cuando el usuario haga clic en la orden *Tx/Rx* del menú *Transmisiones*, se abre el puerto de comunicaciones COM2 y se establecen sus características: para ello se invoca al procedimiento *Abrir_Com* y se le pasa como argumento la cadena *DefCom* que contiene el nombre del puerto y los parámetros bajo los que se desea establecer la comunicación. A continuación, se llama al procedimiento *RecibirCom* para establecer un lazo que mantenga a la aplicación a la espera de recibir datos.

```
Sub TransRec_Click ()
  Abrir_Com DefComS
  If nCid Then
    FormCom.Caption = "Puerto de comunicaciones abierto"
  End If
  RecibirCom
End Sub

Sub Abrir_Com (DefPuertoCom$)
  Dim vr As Integer, Msg As String

  PuertoCom$ = Left$(DefPuertoCom$, InStr(DefPuertoCom$, ":") - 1)
  FormCom.Caption = "Abriendo " + PuertoCom$ + " ..."
  Do
    'Abrir el puerto de comunicaciones. nCid es el ID
    'nCid < 0 indica un error (constantes IE_).

      nCid = OpenComm(PuertoCom$, Tb, Tb)
      If nCid < 0 Then
```



```
    ErrorOpenCom nCid
    If nCid = IE_OPEN Then
      Msg = "Dispositivo COM abierto" + Chr$(13)
      Msg = Msg + "¿Quiere utilizarlo?"
      vr = MsgBox(Msg, 36, "Error en las comunicaciones")
      If vr = 6 Then        'SI: cerrar puerto
        'Cerrar cualquier puerto abierto
        vr = CloseComm(1)
        vr = CloseComm(2)
        'Cerrar otro puerto distinto a los anteriores
        vr = CloseComm(Asc(LpDCB.Id))
      Else                  'NO: finalizar aplicación
        Msg = "Aplicación finalizada"
        MsgBox Msg, 16, "Comunicación abortada"
        End
      End If
    Else
      Msg = "Error al abrir el dispositivo COM" + Chr$(13)
      Msg = Msg + "Revise la conexión y
                   ejecute de nuevo la aplicación"
      vr = MsgBox(Msg, 16, "Error en las comunicaciones")
      End
    End If
  Else
    'Inicializar puerto de comunicaciones PuertoCom$ como:
    '2400,N,8,1,CD0,CS0,DS0,RS,TB2048,RB2048
    If (BuildCommDCB(DefPuertoCom$, LpDCB)) Then
      MsgBox "No se puede construir el DCB", 16
      End
    End If
    'El resto de los parámetros del DCB son puestos manualmente
    LpDCB.Id = Chr$(nCid)
    'Establecer el periodo de tiempo en milisegundos
    'para CD, CS y DS. Un valor 0 representa un tiempo infinito
    'inhabilitando el protocolo sobre esa línea
    LpDCB.RlsTimeOut = 0
    LpDCB.CtsTimeOut = 0
    LpDCB.DsrTimeOut = 0

  'Los siguientes flags de un bit forman el campo ModeControl.
  'Si 1 inhabilita, 0 habilita y viceversa.
  'fBinary = 1        Modo binario. 0 implica EOF (Chr$(26)).
  'fRtsDisabled = 1   Inhabilitar línea request-to-send (RS).
  'fParity = 0        Inhabilitar chequeo de paridad.
  'fOutCtsFlow = 0    Inhabilitar chequeo de clear-to-send.
  'fOutxDsrFlow = 0   Inhabilitar chequeo de data-set-ready (DSR).
  'fDummy = 0 + 0     Dos bit reservados.
  'fDtrDisabled = 1   Inhabilitar línea data-set-ready (DTR).
  'fOutX = 0          Inhabilitar XON/XOFF durante la tr  isión.
  'fInX = 0           Inhabilitar XON/XOFF durante la r    ón.
```

```
      'PeChar = 0  No reemplazar los caracteres de error de paridad con
      '            el carácter contenido en el campo PeChar. 1 reemplaza
      'Null = 0    No descartar caracteres nulos recibidos. 1 descarta
      'RtsCts = 0  La recepción del carácter contenido en el campo
      '            EvtChar no significa un suceso. 1 significa suceso
      'DtrFlow = 0      La línea DTR no es utilizada para recibir. 1 si
      'RtsFlow = 0      La línea RTS no es utilizada para recibir. 1 si
      'Dummy = 0        Reservado
        '
        'Campo ModeControl = 1100 0001 0000 0000
        'En hexadecimal:     &HC     1     0     0

        LpDCB.ModeControl = &HC100
        LpDCB.XonChar = Chr$(0)
        LpDCB.XoffChar = Chr$(0)
        LpDCB.XonLim = 0
        LpDCB.XoffLim = 0
        LpDCB.PeChar = Chr$(0)
        LpDCB.EofChar = Chr$(26)
        LpDCB.EvtChar = Chr$(0)

        FormCom.Caption = "Estableciendo características
                           de  " - PuertoCom$ + " ..."
        vr = SetCommState(LpDCB)
        If vr < 0 Then
          Msg = "Error durante la inicialización del puerto" + Chr$(13)
          Msg = Msg + "Revise la conexión y ejecute de nuevo la
                       aplicación"
          vr = MsgBox(Msg, 16, "Error en las comunicaciones")
          Cerrar_Com
        Else
          'Enfocar la caja de texto
          Text1.SetFocus
        End If
      End If
    Loop While nCid < 0
End Sub

Sub RecibirCom ()
  Dim NumCars As Integer, vr As Integer
  Dim BufferEnt As String * Tb
  Do While nCid
    'Leer datos del puerto
    NumCars = ReadComm(nCid, BufferEnt, Len(BufferEnt))
    If NumCars < 0 Then NumCars = -NumCars
    'Visualizar datos a partir del final de Text1.Text
    If NumCars Then
      Text1.SelStart = Len(Text1.Text)
      Text1.SelText = Left$(BufferEnt, NumCars)
    End If
```



```
    'Restablecer la comunicación si ocurre un error
    ErrorCom 'verificar si ocurrió un error
    'Permitir que otras aplicaciones se ejecuten
    vr = DoEvents()
  Loop
End Sub
```

El procedimiento *ErrorCom* que se presenta a continuación permite visualizar los posibles errores que pueden ocurrir cuando se intenta abrir un dispositivo COM. Este procedimiento es llamado por el procedimiento *Abrir_Com* cuando al invocar a la función **OpenComm** se produce un error.

```
Sub ErrorOpenCom (CódigoError As Integer)
  Dim Msg As String
  Select Case CódigoError
    Case IE_BADID
      Msg = "ID no válido"
    Case IE_BAUDRATE
      Msg = "Velocidad en baudios nó soportada"
    Case IE_BYTESIZE
      Msg = "Tamaño en bytes no válido"
    Case IE_DEFAULT
      Msg = "Error en los parámetros por defecto"
    Case IE_HARDWARE
      Msg = "Hardware no presente"
    Case IE_MEMORY
      Msg = "No es posible asignar colas"
    Case IE_NOPEN
      Msg = "Dispositivo no abierto"
    Case IE_OPEN
      Msg = "Dispositivo abierto"
  End Select
  MsgBox Msg, 48, "Error en las comunicaciones"
End Sub
```

Cuando se realiza una operación de recepción (**ReadComm**) o de transmisión (**WriteComm**) pueden ocurrir errores. Para detectar estos errores, los procedimientos *RecibirCom* y *TransmitirCom*, invocan al procedimiento *ErrorCom* que se presenta a continuación.

```
Sub ErrorCom ()
  Dim CR As String * 1, vr As Integer, Msg As String
  Dim LpStat As COMSTAT
  CR = Chr$(13)    'retorno de carro

  'Verificar si ocurrió un error
  vr = GetCommError(nCid, LpStat)
  If vr <> 0 Then
```

```
If (vr And CE_BREAK) = CE_BREAK Then
  Msg = "Interrupción detectada (break)"
  GoSub VisualizarMensaje
End If
If (vr And CE_CTSTO) = CE_CTSTO Then
  Msg = "Tiempo sobrepasado CTS (Clear-to-send)"
  GoSub VisualizarMensaje
End If

If (vr And CE_DSRTO) = CE_DSRTO Then
  Msg = "Tiempo sobrepasado DSR (Data-set-ready)"
  GoSub VisualizarMensaje
End If

If (vr And CE_DNS) = CE_DNS Then
  Msg = "Dispositivo LPTx no seleccionado"
  GoSub VisualizarMensaje
End If

If (vr And CE_FRAME) = CE_FRAME Then
  Msg = "Error de transmisión (encuadre)"
  GoSub VisualizarMensaje
End If

If (vr And CE_IOE) = CE_IOE Then
  Msg = "Error en dispositivo de E/S" + CR
  Msg = Msg + "intentando establecer comunicación con LPTx"
  GoSub VisualizarMensaje
End If

If (vr And CE_MODE) = CE_MODE Then
  Msg = "Modo solicitado no soportado"
  GoSub VisualizarMensaje
End If

If (vr And CE_OOP) = CE_OOP Then
  Msg = "No hay papel en LPTx"
  GoSub VisualizarMensaje
End If

If (vr And CE_OVERRUN) = CE_OVERRUN Then
  Msg = "Error de sobrescritura en buffer"
  GoSub VisualizarMensaje
End If

If (vr And CE_RTO) = CE_RTO Then
  Msg = "Tiempo sobrepasado en el intento de" + CRS
  Msg = Msg + "establecer comunicación con LPTx"
  GoSub VisualizarMensaje
  If
```



```
    If (vr And CE_RLSDTO) = CE_RLSDTO Then
      Msg = "Tiempo sobrepasado RLSD (Receive-line-signal-detect)"
      GoSub VisualizarMensaje
    End If

    If (vr And CE_RXOVER) = CE_RXOVER Then
      Msg = "Desbordamiento en el buffer de recepción"
      GoSub VisualizarMensaje
    End If

    If (vr And CE_RXPARITY) = CE_RXPARITY Then
      Msg = "Error de paridad"
      GoSub VisualizarMensaje
    End If

    If (vr And CE_TXFULL) = CE_TXFULL Then
      Msg = "Buffer de transmisión lleno"
      GoSub VisualizarMensaje
    End If
  End If
  Exit Sub

VisualizarMensaje:
  MsgBox Msg, 48, "Error en las comunicaciones"
  Return
End Sub
```

La estructura *LpStat* se utiliza como parámetro con la función **GetCommError**, con el fin de que ésta copie en dicha estructura el error ocurrido.

Si ahora el usuario quiere transmitir información, simplemente tiene que escribirla en la caja de texto y pulsar el botón *Enviar texto*. La información escrita es almacenada en la cadena *BufferSal* y enviada al puerto de comunicaciones por el procedimiento *TransmitirCom* que se muestra a continuación.

```
Sub TransmitirCom_Click ()
  Dim Msg As String, NumCars As Integer

  If nCid = 0 Then
    Msg = "Dispositivo COM cerrado" + Chr$(13)
    Msg = Msg + "Para abrirlo ejecute Tx/Rx"
    MsgBox Msg, 48, "Comunicaciones"
  Else
    'Enviar caracteres al buffer de salida
    NumCars = WriteComm(nCid, BufferSal, Len(BufferSal))
    If NumCars < 0 Then NumCars = -NumCars
    ErrorCom 'verificar si ocurrió un error
    'Eliminar los caracteres transmitidos
```

```
    BufferSal = Mid$(BufferSal, NumCars + 1)
  End If
  Text1.SetFocus
End Sub
```

Cuando el usuario quiera interrumpir las comunicaciones, ejecutará la orden *Detener* del menú *Transmisiones*. Esto hace que se ejecute el procedimiento *Detener_Click* que invoca al procedimiento *Cerrar_Com* que se encarga de vaciar los **buffers** de entrada y salida y de finalizar las comunicaciones.

```
Sub Detener_Click ()
  Cerrar_Com
  FormCom.Caption = "Puerto de comunicaciones cerrado"
End Sub

Sub Cerrar_Com ()
  Dim vr As Integer
  If nCid Then
    vr = FlushComm(nCid, 1) 'Vacía el buffer de transmisión
    vr = FlushComm(nCid, 2) 'Vacía el buffer de recepción
    vr = CloseComm(nCid)
  End If
  If vr < 0 Then
    MsgBox "No se puede cerrar el puerto COM: " + Str$(vr), 16
    End
  Else
    nCid = 0 'Poner a cero nCid para detener RecibirCom
  End If
End Sub
```

Para finalizar y salir de la aplicación el usuario puede hacer clic en el menú *Salir*, entonces se ejecuta el procedimiento *Salir_Click*, o en la orden *Cerrar* del menú de control de la forma, con lo que se ejecuta el procedimiento *Form_Unload*. El código para ambos procedimientos se presenta a continuación.

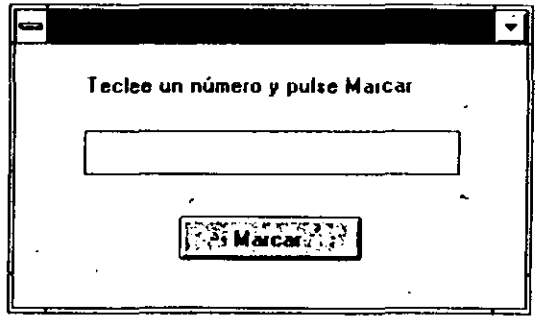
```
Sub Salir_Click ()
  Cerrar_Com
  End
End Sub

Sub Form_Unload (Cancel As Integer)
  Salir_Click
End Sub
```


MARCADOR DE TELÉFONOS

Si dispone de un **modem**, la siguiente aplicación le permitirá marcar cualquier teléfono que previamente teclee en la caja destinada a tal fin. Esta aplicación podría desarrollarse utilizando las funciones de comunicaciones de la API de Windows, pero en este caso prescindiremos de ellas y utilizaremos las sentencias estándar de E/S de Visual Basic.

Inicie una nueva aplicación y cree una forma titulada *Teléfono* con los controles que se indican en la tabla siguiente. Guarde la aplicación con el nombre *teléfono.mak* y la forma con el nombre *teléfono.frm*.

Objeto	Propiedad	Valor
Etiqueta	Caption CtlName	Teclee un número ... Label1
Caja de texto	CtlName Text	Text1 (nada)
Botón	Caption CtlName	Marcar Marcar_Colgar



Antes de probar esta aplicación asegúrese de que el **modem** está conectado y qué puerto utiliza (COM1, COM2, COM3 o COM4); en el ejemplo, hemos supuesto que el **modem** utiliza el puerto COM2.

Cuando ejecute esta aplicación, primeramente teclee en la caja de texto el número que desea marcar y después pulse el botón titulado *Marcar*. En este instante el título del botón cambia a *Colgar*. Cuando la comunicación se establezca, hable. Finalizada la conversación, cuelgue el teléfono y pulse el botón titulado *Colgar*.

Cuando pulse el botón denominado *Marcar_Colgar*, el código verifica el título del mismo es *Marcar* o *Colgar*. Si es *Marcar*, se envía al puerto de comunicaciones una cadena de caracteres formada por un prefijo y el número tecleado, y se cambia el título *Marcar* por *Colgar*. Para un **modem** compatible **Hayes** el prefijo es ATDP si el teléfono es de pulsos o ATDT si el teléfono es de tonos. Si el título es *Colgar* entonces se cierra la comunicación; para ello se envía al puerto la cadena ATH, se cierra el puerto de comunicaciones, y se cambia el título *Colgar* por *Marcar*. El procedimiento *Marcar_Colgar_Click* que se muestra a continuación, recoge todas estas operaciones.

```
Sub Marcar_Colgar_Click ()
  If Marcar_Colgar.Caption = "Marcar" Then
    If Text1.Text <> "" Then
      'pulsos=ATDP, tono=ATDT
      Número$ = "ATDP" + Text1.Text
      Open "COM2" For Output As #1
      Print #1, Número$
      Label1.Caption = "Cuando finalice pulse Colgar"
      Marcar_Colgar.Caption = "Colgar"
    End If
  Else
    Número$ = "ATH"
    Print #1, Número$
    Close #1
    Label1.Caption = "Teclee el número y pulse Marcar"
    Marcar_Colgar.Caption = "Marcar"
  End If
End Sub
```

PARTE 3

Apéndices

- Resumen del lenguaje
- Diferencias entre Visual Basic y Basic
- Códigos de caracteres
- Índice alfabético

APÉNDICE A

RESUMEN DEL LENGUAJE

FUNCIONES, SENTENCIAS, y MÉTODOS

La siguiente tabla presenta las funciones, sentencias, y métodos agrupados por la tarea que desempeñan.

Arrays	
Dim, Global, ReDim, Static	Declarar e inicializar arrays
Erase, ReDim	Reinicializar arrays
LBound, UBound	Obtener los límites de un array
Option Base	Cambiar el límite inferior de un array
Controlar el flujo de un programa	
Do ... Loop, For ... Next, While ... Wend	Realizar bucles o lazos
GoSub ... Return, GoTo, On Error, On ... GoSub, On ... GoTo	Subrutinas, bifurcaciones condicionales y bifurcaciones incondicionales
If ... Then ... Else, Select Case	Tomar decisiones
DoEvents, End, Stop, Unload	Salir de o detener un programa

Convertir	
Chr$	Un valor ASCII a carácter
Asc	Un carácter a su valor ASCII
Str$, Format$	Número a cadena
Val	Cadena a número
CCur, CDbl, CInt, CLng, CSng, Fix, Int	Un tipo de dato numérico a otro
Hex$, Oct$	Un número decimal, a hexadecimal y octal respectivamente
DateSerial, DateValue	Una fecha a un número
Day, Month, Weekday, Year	Un número a una fecha
TimeSerial, TimeValue	Una hora a un número
Hour, Minute, Second	Un número a la hora correspondiente
Now	Un número que representa la fecha y hora actuales
Copiar, cortar y pegar	
Clear, GetData, GetFormat, GetText, SetText, SetData	Utilizar el portapapeles
Date$, Time$	Tomar o poner la fecha o la hora
Timer	Temporizar un proceso
Intercambio dinámico de datos	
LinkSend	Aplicación Visual Basic como servidor
LinkExecute, LinkPoke, LinkRequest	Aplicación Visual Basic como cliente

Gráficos	
QBColor, Point, RGB	Trabajar con colores
Line, Circle, PSet	Dibujar figuras
LoadPicture, SavePicture	Cargar o guardar una imagen
TextHight, TextWidth	Tamaño del texto
Scale	Cambiar el sistema de coordenadas
Print	Escribir texto
Cls	Borrar texto o un gráfico
Manipulación de errores	
On Error, Resume	Interceptar un error durante la ejecución
Err, Erl	N° de error y línea en donde ocurrió
Error$	Mensaje de error
Error	Simular un error
E/S con ficheros	
Open	Abrir o crear un fichero
Close, Reset	Cerrar ficheros
Print #, Put, Write #	Escribir en un fichero
Spc, Tab, Width #	Controlar la apariencia de la salida
Get, Input #, Input$, Line Input #	Leer de un fichero
EOF, FileAttr, FreeFile, Loc, LOF	Tomar información acerca de un fichero
Seek	Posicionarse en un fichero

ChDrive, CurDir$, ChDir, MkDir, RmDir	Trabajo con discos y directorios
Dir$, Kill, Lock ... Unlock, Name	Manipular ficheros
Manipular objetos	
Hide, Show	Ocultar o mostrar una forma
Load, Unload	Cargar o descargar una forma
Move	Mover y ajustar el tamaño de un control
Drag	Arrastrar y dejar caer
SetFocus	Enfocar un control
Refresh	Actualizar un objeto
PrintForm	Imprimir una forma
AddItem, RemoveItem	Añadir o eliminar un elemento en una lista o en un combinado
InputBox$, MsgBox	Visualizar cajas de diálogo
Matemáticas	
Exp, Log, Sqr	e^X, logaritmo y raíz cuadrada
Atn, Cos, Sin, Tan	Arco tangente, coseno, seno y tangente
Fix, Int	Parte entera de un valor
Abs	Valor absoluto
Sgn	Obtener el signo
Randomize, Rnd	Generar números al azar

Imprimir	
EndDoc, NewPage	Finalizar un documento o nueva página
Scale, Spc, Tab, TextHeight, TextWidth	Controlar la apariencia de la salida
Print, PrintForm	Imprimir texto o una forma
Funciones y procedimientos	
Format$	Dar formato a la salida
Space$, String$	Crear una cadena repitiendo un carácter
Asc, Chr$	Valores y caracteres ASCII y ANSI
InStr, Left$, LTrim$, Mid$, Right$, RTrim$	Trabajar con cadenas de caracteres
LCase$, UCase$	Convertir a minúsculas o mayúsculas
LSet, RSet	Justificar una cadena
Len	Longitud de una cadena
Variables y constantes	
Const, Dim, Global, Static	Declarar constantes y variables
Let	Asignar un valor
Def*tipo*	Definir el tipo por defecto (*tipo*: **Int, Lng, Sng, Dbl, Cur, Str**)
Type	Crear un tipo definido por el usuario
Otras	
AppActivate, Shell	Ejecutar otra aplicación
DoEvents	Permitir que se ejecuten otros procesos

SendKeys	Enviar teclas a la ventana activa
Command$	Contiene los argumentos enviados en la línea de órdenes
Environ$	Devuelve una cadena del entorno del sistema operativo
Beep	Pitido

PROPIEDADES

La siguiente tabla presenta las propiedades agrupadas en función del contexto donde se aplican.

Aspecto	
BackColor, ForeColor	Color de fondo y del primer plano
BorderStyle	Estilo del borde
MousePointer	Forma del puntero del ratón
Visible	Ocultar o visualizar un objeto
Intercambio dinámico de datos	
LinkItem, LinkMode, LinkTimeout, LinkTopic	Intercambio dinámico de datos (DDE)
Ficheros	
Drive, FileName, Path, Pattern	Visualizar nombres de ficheros
Archive, Hidden, Normal, ReadOnly, System	Ocultar o mostrar ficheros con esos atributos

Fuentes	
FontCount, Fonts	Fuentes disponibles
FontName, FontSize	Cambiar la fuente actual
Font*estilo*	Estilo de la fuente (*estilo*: **Bold, Italic, Strikethru, Transparent, Underline**)
Formas y controles	
Cancel, Default	Botón por defecto para **Esc** y para **Enter**
Interval	Poner el intervalo de tiempo
Parent	Forma a la que pertenece un control
ScrollBars	Visualizar las barras de desplazamiento
Style	Tipo de combinado
TabIndex, TabStop	Fija el orden seguido por **Tab**
Value	Verifica el estado de un control
CtlName, Index	Nombre e índice de un control
ActiveControl	Retorna el control que está enfocado
List, ListCount, ListIndex, Sorted	Manipular listas
Max, Min, LargeChange, SmallChange	Valores relativos a las barras de desplazamiento
Enabled, Checked	Controlar un elemento de un menú
Enabled	Responder o no a sucesos
ControlBox	Determina si aparece o no el menú de control de una forma

Icon, MaxButton, MinButton, WindowState	Propiedades relacionadas con minimizar o maximizar una forma
FormName	Nombre de la forma
DragIcon, DragMode	Arrastrar y dejar caer
Gráficos	
AutoRedraw	Restaurar una imagen cuando sea preciso
AutoSize	Ajustar el tamaño automáticamente
Image, Picture	Acceder a una imagen
CurrentX, CurrentY, DrawMode, DrawStyle, DrawWidth, FillColor, FillStyle	Dibujar
Imprimir	
Page	Número de página
Tamaño y posición	
Left, Top	Posición de un objeto
Height, Width	Tamaño de un objeto
Texto	
Alignment	Alineación del texto en una etiqueta
AutoSize	Ajustar el tamaño automáticamente
Caption, Text	Título y contenido de un control
SelLength, SelStart, SelText	Texto seleccionado

Ventanas	
hDC	**Handle** del contexto de dispositivo
hWin	**Handle** de la forma

SUCESOS

La siguiente tabla presenta los sucesos que se pueden dar cuando ocurre una determinada acción.

Acción	Sucesos
Cambios en un control	**Change, DropDown, PathChange, PatternChange**
Arrastrar y dejar caer	**DragDrop, DragOver**
Intercambio dinámico de datos	**LinkClose, LinkError, LinkExecute, LinkOpen**
Pulsar una tecla	**KeyPress, KeyDown, KeyUp**
Operaciones con el ratón	**Click, DblClick, MouseDown, MouseUp, MouseMove**
Cambiar el foco (enfocar)	**GotFocus, LostFocus**
Transcurrió el intervalo de tiempo	**Timer**
Trabajando con formas e imágenes	**Paint, Resize, Load, Unload**

OBJETOS

La siguiente tabla presenta una lista de acciones y los objetos que típicamente se utilizan con tales acciones.

Acción	Objeto
Devolver la forma o el control activo durante la ejecución	**Screen**
Elegir un elemento de una lista	**Lista, combinado, grupo de opciones**
Copiar, cortar o pegar	**Clipboard**
Visualizar datos en la ventana inmediata	**Debug**
Visualizar un mapa de bits	**Caja de imagen**
Agrupar controles	**Marco**
Introducir texto	**Caja de texto**
Imprimir texto o gráficos	**Printer**
Desplazamiento de los datos en un objeto	**Barras de desplazamiento horizontal y vertical**
Ejecutar una orden	**Botón, menú**
Seleccionar un directorio	**Lista de directorios**
Seleccionar una unidad de disco	**Lista de unidades de disco**
Activar o desactivar una opción	**Señal, opción**

APÉNDICE B

DIFERENCIAS ENTRE VISUAL BASIC Y BASIC

INTRODUCCIÓN

Visual Basic no es totalmente compatible con otras versiones de BASIC, como GWBASIC y BASICA, o con las versiones posteriores a éstas, QBASIC, Quick Basic y BASIC profesional, lo cual exigirá hacer algunas modificaciones a los programas realizados en estas versiones, para que puedan funcionar en Visual Basic. En lo sucesivo, para referirnos a ambos grupos utilizaremos los términos GWBASIC y QBASIC respectivamente.

CONVERSIÓN DE BASIC A Visual Basic

Cualquier fichero ASCII que contenga código BASIC puede ser importado a Visual Basic ejecutando la orden **Load Text** del menú **Code**.

En Visual Basic, todo el código ejecutable está agrupado en procedimientos (**Sub**) y funciones (**Function**) y la parte del código relativa a declaraciones (**Def***tipo*, **Type**, etc.) se escribe en la sección de declaraciones de la forma o del módulo, y en el módulo global.

Los programas GWBASIC no tienen procedimientos. Por lo tanto, cuando Visual Basic importa un programa GWBASIC, coloca todo el código en la sección de declaraciones de la forma o del módulo actual. A continuación, es tarea del programador adaptar el código y colocarlo en procedimientos y funciones.

Los programas QBASIC tienen procedimientos y funciones. Por lo tanto, cuando Visual Basic importa un programa QBASIC, coloca los procedimientos y las funciones, en procedimientos **Sub** y **Function** a nivel de la forma o del módulo actual, y las declaraciones en la sección de declaraciones correspondiente a dicha forma o módulo. A continuación, es tarea del programador adaptar el código, colocar en procedimientos y funciones el código que aún no lo esté, y colocar en el módulo global las declaraciones que correspondan.

PALABRAS CLAVE NO SOPORTADAS

Visual Basic no soporta las siguientes palabras clave:

BLOAD	BSAVE	CALL ABSOLUTE	CALLS	CHAIN
CLEAR	COLOR	COM	CSRLIN	CVD
CVDMBF	CVI	CVS	CVSMBF	DATA
DEF FN	DEF SEG	DRAW	ERDEV	ERDEVS
FIELD	FILES	FRE	INKEYS	INP
IOCTL	IOCTL$	KEY	LOCATE	LPOS
LPRINT	MKD$	MKI$	MKL$	MKS$
MKSMBF$	ON COM	ON KEY	ON PLAY	ON STRIG
ON TIMER	OUT	PAINT	PALETTE	PCOPY
PEEK	PEN	PLAY	PMAP	POKE
POS	PRESERVE	PRESET	RESTORE	RUN
SADD	SCREEN	SETMEM	SLEEP	SOUND
STICK	STRIG	SWAP	TROFF	TRON
USING$	VARPTR	VARPTR$	VARSEG	VIEW
WAIT	WIDTH LPRINT	WINDOW		

En la mayoría de los casos, una palabra clave no es soportada porque Visual Basic aporta los medios necesarios para lograr el mismo resultado. Por ejemplo, la sentencia LPRINT no es soportada porque Visual Basic provee el objeto **Printer** que controla toda información enviada a la impresora. En otros casos, una palabra clave no es soportada porque involucra operaciones de bajo nivel que entran en conflicto con el entorno Windows.

En Visual Basic, cualquier referencia a una línea etiquetada (**GoSub** o **GoTo**) debe ser local al procedimiento. No obstante, una rutina invocada con **GoSub** puede ser convertida en un procedimiento. Asimismo, las funciones definidas con la sentencia DEF FN no soportada, tienen que ser convertidas en funciones (**Function**).

ÁMBITO DE LAS VARIABLES

Se entiende por ámbito de una variable, el espacio de la aplicación donde la variable es visible y por lo tanto se puede utilizar (para más información vea el Capítulo 3).

En GWBASIC, todas las variables son compartidas por todo el programa; en Visual Basic las variables son locales por defecto. Por lo tanto, cuando se importe un programa GWBASIC se pueden dejar todas las variables locales por defecto, y declarar en la sección de declaraciones de la forma, del módulo, o del módulo global, donde corresponda, aquellas que sean compartidas por varios procedimientos.

Las variables locales en un programa QBASIC se trasladan sin cambios a Visual Basic. Las sentencia COMMON SHARED en QBASIC hay que reemplazarla por la declaración **Global** en el módulo global, y DIM SHARED por la declaración DIM a nivel de la forma o del módulo.

En resumen, una vez importado un programa, para evitar errores trataremos de conservar el ámbito de cada una de las variables siempre que sea posible. Cuando esto no sea posible, puede ser necesario cambiar el nombre de algunas variables para evitar conflictos. También habrá que tener en cuenta que el nombre de una variable no coincida con el nombre de un objeto o de una propiedad.

CONSIDERACIONES GENERALES

La sentencia **Declare** solamente es válida para rutinas DLL y para funciones (**Function**) sin argumentos, que son llamadas sin utilizar paréntesis. Esto es, una llamada a una función, **Function**, tiene que utilizar paréntesis aunque la función no tenga argumentos. Por ejemplo,

```
vg = MiFunción ()
```

Para poder omitir los paréntesis es necesario incluir una sentencia **Declare** en la sección de declaraciones de la forma o del módulo actual.

Una rutina para manipulación de errores (**On Error ...**) tiene que ser siempre local al procedimiento. La rutina y la sentencia **On Error** tienen que estar en el mismo procedimiento.

La sentencia PRINT de Basic tiene el mismo efecto que la sentencia **Print** en Visual Basic con la diferencia que ésta escribe en la forma actual. Para escribir en otro lugar, por ejemplo en la caja de imagen *Picture1* hay que utilizar la sintaxis,

```
Picture1.Print ...
```

Otras alternativas para visualizar datos son la sentencia y la función **MsgBox**.

La sentencia INPUT generalmente la reemplazaremos con la función **InputBox$**. Otra solución es crear una caja de texto y utilizar su propiedad **Text**.

Visual Basic no soporta la sentencia INKEY. Para reemplazarla utilizaremos un procedimiento conducido por el suceso **KeyPress**.

La sentencia **Def***tipo* sólo puede utilizarse en la sección de declaraciones de la forma o del módulo, y en el módulo global. Esta sentencia tiene que escribirse antes de cualquier sentencia **Declare**.

No se puede definir un array implícitamente. Los arrays tienen que ser declarados explícitamente utilizando la sentencia **Dim** o **ReDim**. Si en la sentencia **Dim** se especifican las dimensiones, se crea un array de longitud fija. Por ejemplo,

```
Dim MiArray(99)
```

Esta sentencia puede aparecer a cualquier nivel. No obstante, cuando aparece dentro de un procedimiento, tiene que especificarse antes de cualquier sentencia ejecutable. Si en la sentencia **Dim** se omiten las dimensiones, se crea un array dinámico. Posteriormente, utilizaremos la sentencia **ReDim** para asignar espacio. Por ejemplo,

```
Dim MiArray()
...
ReDim MiArray(99)
```

También se puede crear un array utilizando directamente la sentencia **ReDim**. Como esta sentencia es ejecutable, sólo puede aparecer en cualquier parte dentro de un procedimiento.

La sentencia **Option Base** sólo puede utilizarse en la sección de declaraciones de la forma o del módulo, y en el módulo global. Esta sentencia tiene que especificarse antes de cualquier referencia a un array.

La sentencia **Input$** (*n, id_fichero*) devuelve un *EOF* (fin de fichero) cuando intenta leer de un fichero en el que quedan menos de *n* caracteres. Una solución a este problema puede plantearse a través de la sentencia **On Error**. Otra solución es hacer que *n* valga 1, como en el ejemplo que se presenta a continuación.

```
Open FicheroS For Input As #1
carS = Input$(1, #1)
Do While Not EOF(1)
  If Asc(carS) <> 10 Then Print carS;
  carS = Input$(1, #1)
Loop
Close
```

Visual Basic acepta la sentencia **Return**, pero no **Return** *id_línea*.

En la sentencia **Width #** *id_fichero, n*, si *n* vale *0* indica que el fichero o dispositivo no tiene límite de ancho (para esto, otras versiones de Basic utilizaban *n=255*). Un valor de *n>0* indica donde finaliza una línea. Visual Basic no soporta las sentencias:

WIDTH *columnas_pantalla, líneas_pantalla*
WIDTH *dispositivo, ancho*

Otras consideraciones de interés son:

- Para referenciar una propiedad, un método o un miembro de un tipo definido por el usuario, se utiliza como separador el punto. Por lo tanto no utilice este carácter en la declaración de variables o constantes.

- Visual Basic soporta líneas de hasta 256 caracteres, pero no tiene un carácter de continuación de línea.

- Visual Basic no permite que dos variables declaradas al mismo nivel tengan el mismo nombre, aunque tengan diferentes tipos. Por ejemplo *a%* y *aS* no pueden existir al mismo nivel de declaración.

APÉNDICE C

CÓDIGOS DE CARACTERES

UTILIZACIÓN DE CARACTERES ANSI CON WINDOWS

Una tabla de códigos es un juego de caracteres donde cada uno tiene asignado un número utilizado para su representación interna.

Windows utiliza el juego de caracteres ANSI. Para escribir un carácter ANSI que no esté en el teclado:

1. Localice en la tabla que se muestra en la página siguiente el carácter ANSI que necesite y observe su código numérico.
2. Pulse la tecla **Bloq Num** (Num Lock) para activar el teclado numérico.
3. Mantenga pulsada la tecla **Alt** y utilice el teclado numérico para pulsar el 0 y a continuación las teclas correspondientes al código del carácter.

Por ejemplo, para escribir el carácter ± bajo el entorno Windows, mantenga pulsada la tecla **Alt** mientras escribe 0177 en el teclado numérico.

JUEGO DE CARACTERES ANSI

DEC	CAR	DEC	CAR	DEC	CAR	DEC	CAR
33	!	89	Y	146	'	202	Ê
34	"	90	Z	147	"	203	Ë
35	#	91	[	148	"	204	Ì
36	$	92	\	149	•	205	Í
37	%	93	]	150	–	206	Î
38	&	94	^	151	—	207	Ï
39	'	96	`	152	˜	208	Ð
40	(	97	a	153	™	209	Ñ
41	)	98	b	154	š	210	Ò
42	*	99	c	155	›	211	Ó
43	+	100	d	156	œ	212	Ô
44	,	101	e	157	▯	213	Õ
45	-	102	f	157	▯	214	Ö
46	.	103	g	159	Ÿ	215	×
47	/	104	h	160		216	Ø
48	0	105	i	161	¡	217	Ù
49	1	106	j	162	¢	218	Ú
50	2	107	k	163	£	219	Û
51	3	108	l	164	¤	220	Ü
52	4	109	m	165	¥	221	Ý
53	5	110	n	166	¦	222	Þ
54	6	111	o	167	§	223	ß
55	7	112	p	168	¨	224	à
56	8	113	q	169	©	225	á
57	9	114	r	170	ª	226	â
58	:	115	s	171	«	227	ã
59	;	116	t	172	¬	228	ä
60	<	117	u	173	-	229	å
61	=	118	v	174	®	230	æ
62	>	119	w	175	¯	231	ç
63	?	120	x	176	°	232	è
64	@	121	y	177	±	233	é
65	A	122	z	178	²	234	ê
66	B	123	{	179	³	235	ë
67	C	124	|	180	´	236	ì
68	D	125	}	181	µ	237	í
69	E	126	~	182	¶	238	î
70	F	127	▯	183	·	239	ï
71	G	128	▯	184	¸	240	ð
72	H	129	▯	185	¹	241	ñ
73	I	130	▯	186	º	242	ò
74	J	131	▯	187	»	243	ó
75	K	132	▯	188	¼	244	ô
76	L	133	▯	189	½	245	õ
77	M	134	▯	190	¾	246	ö
78	N	135	▯	191	¿	247	÷
79	O	136	▯	192	À	248	ø
80	P	137	▯	193	Á	249	ù
81	Q	138	▯	194	Â	250	ú
82	R	139	▯	195	Ã	251	û
83	S	140	▯	196	Ä	252	ü
84	T	141	▯	197	Å	253	ý
85	U	142	▯	198	Æ	254	þ
86	V	143	▯	199	Ç	255	ÿ
87	W	144	▯	200	È		
88	X	145		201	É		

UTILIZACIÓN DE CARACTERES ASCII

Bajo MS-DOS y fuera del entorno Windows se utiliza el juego de caracteres ASCII. Para escribir un carácter ASCII que no esté en el teclado:

1. Busque el carácter en la tabla de códigos que coincida con la tabla activa. Utilice la orden **chcp** para saber qué tabla de códigos está activa.
2. Mantenga pulsada la tecla **Alt** y utilice el teclado numérico para pulsar las teclas correspondientes al número del carácter que desee.

Por ejemplo, si está utilizando la tabla de códigos 850, para escribir el carácter π mantenga pulsada la tecla **Alt** mientras escribe 227 en el teclado numérico.

JUEGO DE CARACTERES ASCII

VALOR DECIMAL	VALOR HEXADECIMAL	CONTROL CARACT	CARACT	VALOR DECIMAL	VALOR HEXADECIMAL	CARACT	VALOR DECIMAL	VALOR HEXADECIMAL	CARACT	VALOR DECIMAL	VALOR HEXADECIMAL	CARACT	VALOR DECIMAL	VALOR HEXADECIMAL	CARACT
000	00	NUL		043	2B	+	086	56	V	129	81	ü	172	AC	¼	215	D7	÷
001	01	SOH	☺	044	2C	,	087	57	W	130	82	é	173	AD	¡	216	D8	°
002	02	STX	☻	045	2D	-	088	58	X	131	83	â	174	AE	«	217	D9	∙
003	03	ETX	♥	046	2E	.	089	59	Y	132	84	ä	175	AF	»	218	DA	·
004	04	EOT	♦	047	2F	/	090	5A	Z	133	85	à	176	B0	▒	219	DB	√
005	05	ENQ	♣	048	30	0	091	5B	[	134	86	å	177	B1	▒	220	DC	ⁿ
006	06	ACK	♠	049	31	1	092	5C	\	135	87	ç	178	B2	▓	221	DD	²
007	07	BEL	•	050	32	2	093	5D	]	136	88	ê	179	B3	│	222	DE	■
008	08	BS	◘	051	33	3	094	5E	^	137	89	ë	180	B4	┤	223	DF	
009	09	HT	○	052	34	4	095	5F	_	138	8A	è	181	B5	╡	224	E0	Ó
010	0A	LF	◙	053	35	5	096	60	`	139	8B	ï	182	B6	╢	225	E1	ß
011	0B	VT	♂	054	36	6	097	61	a	140	8C	î	183	B7	╖	226	E2	Ô
012	0C	FF	♀	055	37	7	098	62	b	141	8D	ì	184	B8	╕	227	E3	Ò
013	0D	CR	♪	056	38	8	099	63	c	142	8E	Ä	185	B9	╣	228	E4	õ
014	0E	SO	♫	057	39	9	100	64	d	143	8F	Å	186	BA	║	229	E5	Õ
015	0F	SI	☼	058	3A	:	101	65	e	144	90	É	187	BB	╗	230	E6	µ
016	10	DLE	►	059	3B	;	102	66	f	145	91	æ	188	BC	╝	231	E7	þ
017	11	DC1	◄	060	3C	<	103	67	g	146	92	Æ	189	BD	╜	232	E8	Þ
018	12	DC2	↕	061	3D	=	104	68	h	147	93	ô	190	BE	╛	233	E9	Ú
019	13	DC3	‼	062	3E	>	105	69	i	148	94	ö	191	BF	┐	234	EA	Û
020	14	DC4	¶	063	3F	?	106	6A	j	149	95	ò	192	C0	└	235	EB	Ù
021	15	NAK	§	064	40	@	107	6B	k	150	96	û	193	C1	┴	236	EC	ý
022	16	SYN	▬	065	41	A	108	6C	l	151	97	ù	194	C2	┬	237	ED	Ý
023	17	ETB	↨	066	42	B	109	6D	m	152	98	ÿ	195	C3	├	238	EE	¯
024	18	CAN	↑	067	43	C	110	6E	n	153	99	Ö	196	C4	─	239	EF	´
025	19	EM	↓	068	44	D	111	6F	o	154	9A	Ü	197	C5	┼	240	F0	≡
026	1A	SUB	→	069	45	E	112	70	p	155	9B	¢	198	C6	╞	241	F1	±
027	1B	ESC	←	070	46	F	113	71	q	156	9C	£	199	C7	╟	242	F2	‗
028	1C	FS	∟	071	47	G	114	72	r	157	9D	¥	200	C8	╚	243	F3	¾
029	1D	GS	↔	072	48	H	115	73	s	158	9E	₧	201	C9	╔	244	F4	¶
030	1E	RS	▲	073	49	I	116	74	t	159	9F	ƒ	202	CA	╩	245	F5	§
031	1F	US	▼	074	4A	J	117	75	u	160	A0	á	203	CB	╦	246	F6	÷
032	20	SP	Space	075	4B	K	118	76	v	161	A1	í	204	CC	╠	247	F7	¸
033	21		!	076	4C	L	119	77	w	162	A2	ó	205	CD	═	248	F8	°
034	22		"	077	4D	M	120	78	x	163	A3	ú	206	CE	╬	249	F9	¨
035	23		#	078	4E	N	121	79	y	164	A4	ñ	207	CF	╧	250	FA	·
036	24		$	079	4F	O	122	7A	z	165	A5	Ñ	208	D0	╨	251	FB	¹
037	25		%	080	50	P	123	7B	{	166	A6	ª	209	D1	╤	252	FC	³
038	26		&	081	51	Q	124	7C	|	167	A7	º	210	D2	╥	253	FD	²
039	27		'	082	52	R	125	7D	}	168	A8	¿	211	D3	╙	254	FE	■
040	28		(	083	53	S	126	7E	~	169	A9	⌐	212	D4	╘	255	FF	
041	29		)	084	54	T	127	7F		170	AA	¬	213	D5	╒			
042	2A		*	085	55	U	128	80	Ç	171	AB	½	214	D6	╓			

CÓDIGOS EXTENDIDOS

Segundo código	Significado
3	NUL (null character)
15	Shift Tab (— < + +)
16-25	Alt-Q/W/E/R/T/Y/U/I/O/P
30-38	Alt-A/S/D/F/G/H/I/J/K/L
44-50	Alt-Z/X/C/V/B/N/M
59-68	Keys F1-F10 (disabled as softkeys)
71	Home
72	Up arrow
73	PgUp
75	Left arrow
77	Right arrow
79	End
80	Down arrow
81	PgDn
82	Ins
83	Del
84-93	F11-F20 (Shift-F1 to Shift-F10)
94-103	F21-F30 (Ctrl-F1 through F10)
104-113	F31-F40 (Alt-F1 through F10)
114	Ctrl-PrtSc
115	Ctrl-Left arrow
116	Ctrl-Right arrow
117	Ctrl-End
118	Ctrl-PgDn
119	Ctrl-Home
120-131	Alt-1/2/3/4/5/6/7/8/9/0/—/=
132	Ctrl-PgUp
133	F11
134	F12
135	Shift-F11
136	Shift-F12
137	Ctrl-F11
138	Ctrl-F12
139	Alt-F11
140	Alt-F12

CÓDIGOS DEL TECLADO

Tecla	Código en Hex	Tecla	Código en Hex
Esc	01	*Left/Right orrow*	0F
!1	02	*Q*	10
@2	03	*W*	11
#3	04	*E*	12
$4	05	*R*	13
%5	06	*T*	14
^6	07	*Y*	15
&7	08	*U*	16
**8*	09	*I*	17
(9	0A	*O*	18
)0	0B	*P*	19
_=	0C	*{[*	1A
+ =	0D	*}]*	1B
Backspace	0E	*Enter*	1C
Ctrl	1D	*|*	2B
A	1E	*Z*	2C
S	1F	*X*	2D
D	20	*C*	2E
F	21	*V*	2F
G	22	*B*	30
H	23	*N*	31
J	24	*M*	32
K	25	*<,*	33
L	26	*>.*	34
:;	27	*?/*	35
"'	28	*RightShift*	36
~`	29	*PrtScr**	37
LeftShift	2A	*Alt*	38
Spacebar	39	*7Home*	47
Caps Lock	3A	*8Up arrow*	48
F1	3B	*9PgUp*	49
F2	3C	*—*	4A
F3	3D	*4Left arrow*	4B
F4	3E	*5*	4C
F5	3F	*6Right arrow*	4D
F6	40	*+*	4E
F7	41	*1End*	4F
F8	42	*2Down arrow*	50
F9	43	*3PgDn*	51
F10	44	*0Ins*	52
F11	D9	*Del*	53
F12	DA		
Num Lock	45		
Scroll Lock	46		

APÉNDICE D

ÍNDICE ALFABÉTICO

A

abrir fichero, 164, 177
Accelerator, 98
Access, 178
activar otra aplicación, 322
ActivateControl, 126; 320
ActivateForm, 126
ActiveControl, 320
AddItem, 142
algoritmos Hash, 267
Alias, 327
ámbito
  de un procedimiento, 48
  de una variable, 38
ancho de línea, 208
animación, 191
  de una imagen, 214
ANSI, 415
Any, 327
añadir un elemento a una lista, 142
API, 325
aplicación, 3; 6
  crear, 5; 19
  depurar, 31
  ejecutar, 31
  guardar, 30
aplicaciones Windows, 112
AppActivate, 322
Append, 165
argumentos
  de los sucesos del ratón, 219
  por referencia y por valor, 51
arrancar Visual Basic, 14
arrastrar, 238
arrastre
  de una imagen, 242
  manual, 248
Arrays, 329
  de controles, 56; 78
  de estructuras, 122
  de registros, 141; 176
  de variables, 55
  dinámicos, 56
  estáticos, 55
  hash, 267
Asc, 66
ASCII, 417
AutoRedraw, 200; 203; 206
AutoSize, 191
ayuda, 12

B

BackColor, 71
barra
  de menús, 16
  de propiedades, 7; 24
  de título, 16
barras de desplazamiento, 16; 22; 150
BorderStyle, 142
borrar un control, 24
botón
  del ratón más tecla, 222
  por defecto, 72
  pulsado, 220
botones, 22; 72
Break, 259
BuildCommDCB, 377

BYTE, tipo, 333
ByVal, 52; 327

C

caja de texto multilínea, 99
caja de texto, 22
calculadora, 83
campo, 139
Cancel, 339
Caption, 63
cargar una aplicación una sola vez, 361
Change, 32; 152; 168
Checked, 98
Circle, 211
Clear, 102
Click, 28; 171
cliente, 289; 291; 296
Clipboard, 101
Close, 165
CloseComm, 378
CloseSound, 366
Cls, 200; 204
código
  unir a los objetos, 27
  visualizar, 7
color, 215
colorear figuras, 213
colores, 11; 153
COM, 381; 388
combinado, 22; 139; 147
comentario, 35
CommandS, 14
comunicaciones, 375
CONSTANT.TXT, 112; 297
constante de caracteres, 36
constante, 36
contexto de dispositivo, 331
control
  añadir, 195
  añadir, eliminar, 81
  borrar, 24
  crear, 23
  mover, 23
  mover, modificar, 193
  tipo Control, 52
ControlBox, 246
controles
  crear en ejecución, 81
  Drive, Dir y File, 168
controles, 7
  dibujar, 4; 21
coordenadas
  del puntero del ratón, 223
  relativas, 207
Copiar, 317
CountVoiceNotes, 375
crear
  control, 23
  controles en ejecución, 81
  un fichero ejecutable, 32
  un módulo, 39; 75; 144
  un procedimiento general, 45
  una aplicación, 5; 19; 62
  una forma, 4
CreateCompatibleDC, 351
CtlName, 63
CurDirS, 256
Currency, 38
CurrentX, 199
CurrentY, 199
cursor del ratón, 251

D

DDE, 289
de acceso, 268
Debug.Print, 261
declarar una función de una DLL, 326
Declare, 50; 326
Default, 72; 80; 169
DeleteDC, 355
depurar una aplicación, 31; 259
dibujar los controles, 4; 21
dibujar
  un círculo, una elipse o un arco, 211
  un punto, 204
  una caja, 209
  una línea, 207
dibujos, 224
Dim, 37; 412
DirS, 256
directorio, seleccionar, 171
disco, seleccionar, 170
diseño, 259
DLL, 50; 325
Do ... Loop, 46
DoEvents, 305; 314
DOS, ejecutar utilidad, 359
Drag, 242
DragDrop, 225; 248; 259
DragItem, 238; 246
DragMode, 238
DragOver, 229; 248
DrawMode, 209
DrawStyle, 208
DrawWidth, 205; 208
Drive, 168

E

E/S de datos, 131
ejecución paso a paso, 260
ejecución, 259
  inicio, 365
  de una aplicación, 31
  de órdenes del sistema, 75
  de un programa Shell, 153
ejecutar, 20
eliminación de elementos, 271
eliminar un elemento de una lista, 145
Else, 42
ElseIf ... TypeOf, 53
EM_GETLINECOUNT, 343
EM_GETMODIFY, 343
EM_LINEFROMCHAR, 343
EM_LINEINDEX, 343
EM_SETMODIFY, 343
EM_UNDO, 343
Enabled, 98; 134; 136
End Function, 49
EndDoc, 358
enfocar un objeto, 126
enlace caliente, 291
enlace frío o caliente, 302
enlace frío, 301
entorno operativo, 305
enviar teclas, 322
EOF, 175
Err, 258
ErrorS, 258
Error, 259
errores, 255; 258
  código de, 165
  DDE, 311
  en comunicaciones, 384
  manipulación de, 254
ES_PASSWORD, 347
ES_SETPASSWORDCHAR, 347
Escape, 357
escribir fichero, 164; 177
escribir texto, 199
estilo, 208
estructuras de control, 42
estructuras, 58; 330
etiqueta, 22; 63
EXE, crear fichero, 32
Exit Do, 47
Exit For, 45
Exit Function, 49
Exit Sub, 50
expresión de Boole, 43

F

fichero
  abrir, 177
  abrir, escribir, cerrar, 164
  aleatorio, 162; 176
  binario, 162; 187
  borrar, 178
  cambiar nombre, 185
  cerrar, 164
  de datos, 161
  ejecutable, 32
  escribir, 177
  indexado, 266; 271
  leer, 180
  secuencial, 162
  seleccionar, 172
ficheros largos, 175
FileName, 168
FillColor, 213
FillStyle, 213
final del fichero, 175
finalizar la ejecución, 21
foco, 126
FontName, 107
For ... Next, 45
ForeColor, 115
Form, tipo, 52
Form_Load, 110
forma, 7
  añadir, 124
  cargar, 110
  crear, 4
  mover y ajustar, 20
  ocultar, 110
  sucesos, 29
  tipo Forth, 52
  visualizar, 7; 110
formas y controles, 333
formas
  manipular, 111
  referencias a, 111
FormatS, 69
Fuentes, 107; 286
FlushComm, 394
función
  Asc, 66
  CurDirS, 256
  de acceso, 268
  DirS, 256
  DoEvents, 305; 314
  EOF, 175
  Err, 258
  ErrorS, 258
  FormatS, 69
  HexS, 137

InputS, 175
InputBoxS, 129; 132
InStr, 138; 202
Len, 105
LoadPicture, 190
Loc, 188
LOF, 188
LtrimS, 128
MidS, 75
MsgBox, 132; 178
Now, 70
OctS, 137
QBColor, 155
RGB, 154
RightS, 128
RtrimS, 145
Shell, 75; 152
StrS, 42; 69
Tab, 211
TimeS, 116
UCaseS, 66
Val, 42
Valor, 75
funciones de la API de Windows, 326
funciones, 48; 399
Function, 48; 411

G

Get, 180; 187
GetActiveWindow, 363
GetCommError, 377
GetFocus, 343
GetFormat, 321
GetModuleHandle, 365
GetModuleUsage, 365
GetNumTasks, 361
GetParent, 353
GetTempFileName, 333
GetText, 101; 317
GetWindowLong, 346
Global, 40
global, módulo, 57
GotFocus, 126
gráficos, 189; 224
borrar, 204
en un DDE, 310
persistentes, 203
guardar la aplicación, 30
GWL_STYLE, 346

H

Hash,
abierto, 268
con overflow, 270
eliminación de un elemento, 271
hDC, 331
Height, 193; 197
HexS, 137
Hide, 110
hWnd, 331

I

Icon, 112; 246
iconos, 109; 191
asociar, 112
If ... Then ... Else ..., 42
If ... TypeOf, 53
If, 53
Image, 322
imagen, 22
añadir, 109; 189
tipos, 189
inmediata, 10; 32; 261
imprimir resultados, 285
Index, 56; 57; 79; 86; 99
indexado, 266
inicio de la ejecución, 365
InputS, 172; 175; 413
InputBoxS, 129; 132
instalar Visual Basic, 13
InStr, 138; 202
interfaz, 61
Interval, 114
IsWindow, 363

K

KeyDown, 67; 105
KeyPress, 66; 116; 237
KeyUp, 67; 105
Kill, 178

L

LargeChange, 151
LB_RESETCONTENT, 346
leer fichero, 172; 180
Left y Top, 82
Left, 193
Len, 105
Lib, 327
librería dinámica, 325
Line Input, 175
Line, 207; 209

LinkClose, 308; 309
LinkError, 308; 309
LinkExecute, 309
LinkExecute, 310
LinkItem, 290
LinkMode, 291
LinkOpen, 307; 308
LinkPoke, 302
LinkRequest, 291; 301
LinkSend, 310
LinkTimeout, 292
LinkTopic, 290
List y ListIndex, 171
List, 139; 239
lista, 22; 138
de directorios, 23; 168
de ficheros, 23; 168
de unidades de disco, 22; 168
ListCount, 139; 238
ListIndex, 146; 237
llamada
a un procedimiento, 50
a una función de una DLL, 328
a una función, 49
Load, 29; 81; 111; 123
LoadPicture, 190
Loc, 188
LOF, 188
longitud de búsqueda, 267
longitud del fichero, 174
Loop, 46
LostFocus, 126
LtrimS, 128

M

Main, 365
manipular formas, 111
marco, 22; 152; 194
Max, 150
MaxButton, 142
maximizar, 16
mensajes, 132; 334
enviar, 334; 345
Menú de control, 16
Menu Design Window, 96
menús, 10; 95
añadir órdenes, 120
borrar órdenes, 128
cambiar, 118
líneas de separación, 97
modificar, 113
orden, 97
propiedades, 98
métodos, 21; 399
AddItem, 142
Circle, 211
Clear, 102
Cls, 204
Drag, 232
EndDoc, 358
GetFormat, 321
GetText, 101; 317
hash abierto, 268
hash con overflow, 270
Hide, 110
Line, 207; 209
LinkExecute, 310
LinkPoke, 302
LinkRequest, 291; 301
LinkSend, 310
Move, 193; 196
NewPage, 286; 357
Point, 155; 217
Print, 107; 199; 237
PrintForm, 354
Pset, 204
RemoveItem, 145
Scale, 198
SetData, 322
SetFocus, 126; 255
SetText, 101; 318
Show, 110; 143
TextHeight, 200
TextWidth, 200
MidS, función y sentencia, 75
Min, 150
MinButton, 142
minimizar, 16
modem compatible Hayes, 396
modificadores de tamaño, 23
modificar un módulo, 144
modo
Append, 165
Output, 165
Random, 177
módulo, 8; 143
crear, 39; 75
global, 30; 57
MouseDown, 219
MouseMove, 219
MouseUp, 219
Move, 193; 196
mover un control, 23
MsgBox, 129; 132
MsgBox, 132; 178
MultiLine, 99
múltiples formas, 111
música, 369

N

Name, 185
NEWFRAME, 357
NewPage, 286; 357
nombre
  de un control, 23
  de un fichero, 173
  de una variable, 36
Now, 70

O

objetos, 3
  general, 29
  Printer, 168; 285; 286; 358
  Screen, 126; 320
objetos, 407
Oct$, 137
ocultar una forma, 110
On Error, 165; 257; 411
opción, 22; 135
opciones de la orden vb, 14
Open, 161; 177
OpenComm, 377
OpenSound, 366
operadores, 30
Option Base, 412
orden de un menú, 97
origen de coordenadas, 196
Output, 165

P

Paint, 203
palabra reservada, 37
panel
  de control, 155
  de dibujo, 226
  de utilidades, 21
papelera, ejemplo, 243
Path, 168
PathChange, 168
Pattern, 168
pausa, 259
Pegar Vínculos, 317
Picture, 109; 246
PLAY, 371
Point, 155; 217
portapapeles, 101
Print, 107, 164, 199; 237; 261
Printer, 285
  método, 286
  propiedades, 286
PrintForm, 354
procedimiento, 8
  conducido por un suceso, 47
  Form_Load, 110
  general, 47
    crear, 48
    editar, 75
  cómo trabaja, 68
Procedimientos, 50
propiedad, 27
  Accelerator, 98
  ActivateControl, 126
  ActivateForm, 126
  ActiveControl, 320
  AutoRedraw, 200; 203; 206
  AutoSize, 191
  BackColor, 71
  BorderStyle, 142
  Caption, 63
  Checked, 98
  ControlBox, 246
  CtlName, 63
  CurrentX, 199
  CurrentY, 199
  Default, 72; 80; 169
  Deftipo, 57; 411
  DragIcon, 238; 246
  DragMode, 238
  DrawMode, 209
  DrawStyle, 208
  DrawWidth, 205; 208
  Drive, 168
  Enabled, 98; 134; 136
  FileName, 168
  FillColor, 213
  FillStyle, 213
  ForeColor, 115
  FontName, 107
  hDC, 331
  Height, 193; 197
  hWnd, 331
  Icon, 112; 246
  Image, 322
  Index, 56; 79; 86; 99
  Interval, 114
  LargeChange, 151
  Left, 193
  LinkItem, 290
  LinkMode, 291
  LinkTimeout, 292
  LinkTopic, 290
  List, 139; 239
  ListCount, 139; 238
  ListIndex, 136; 237
  Max, 150
  MaxButton, 142
  Min, 150
  MinButton, 142
  MultiLine, 99
  Path, 168
  Pattern, 168
  Picture, 109; 246
  ScaleHeight, 197
  ScaleLeft, 198
  ScaleMode, 197
  ScaleTop, 198
  ScaleWidth, 197
  ScrollBars, 99
  SelLenght, 101
  SelStart, 101
  SelText, 101
  SmallChange, 151
  Sorted, 130
  Style, 147
  TabIndex, 255; 348
  TabStop, 72
  Tag, 246
  Text, 68; 176
  Top, 193
  valor de una propiedad, 25
  Value, 134; 136; 150
  Visible, 81; 99
  Width, 193; 197
propiedades 7; 24; 41; 332; 401
  de las cajas de texto, 101
  de listas y combinados, 148
  de un menú, 98
  Left y Top, 82
  List y ListIndex, 171
  Width y Height, 82
proyecto, 6
Pset, 204
puerto serie, 375
puntero nulo, 330
puntero, 22
punto de parada, 260
Put, 177; 187

Q

QBColor, 155

R

random, 177
ratón, 219
ReadComm, 377
recursividad, 52
ReDim, 56; 412
referencia, argumento por 51
registro, 58; 139
rejilla, 20
  eliminar, 64
RemoveItem, 145
Resume, 166
Return, 413
RGB, 154
Right$, 128
rs232, 378; 384
Rtrim$, 135

S

Scale, 198
ScaleHeight, 197
ScaleLeft, 198
ScaleMode, 197
ScaleTop, 198
ScaleWidth, 197
Screen, 351; 320
Screen.MousePointer, 234; 251
ScrollBars, 99
ScrollWindow, 382
Seek, 187
seleccionar
  texto, 100
  un objeto, 25
  y arrastrar, 238
Select Case, 43
SelectObject, 354
SelLenght, 101
SelStart, 101
SelText, 101
SendKeys, 323
SendMessage, 334; 343
sentencia, 41
  AppActivate, 322
  Debug.Print, 261
  Error, 259
  Get, 180; 187
  Kill, 178
  Line Input, 175
  Load, 111; 123
  MsgBox, 129; 132
  Name, 185
  On Error, 257
  Print, 261
  Put, 177; 187
  Seek, 187
  SendKeys, 323
  Unload, 111; 128
sentencias, 399
señal, 22; 134
separación, 97
separador decimal, 74

servidor, 289; 293; 302
SetCommState, 377
SetData, 322
SetFocus, 126; 255
SetFocusAPI, 336
SetText, 101; 318
SetVoiceAccent, 366
SetVoiceNote, 367
SetVoiceQueueSize, 366
SetVoiceSound, 367
SetWindowLong, 336
Shell, 75; 152
Show, 110; 143
sistema de coordenadas, 197
sistema de ficheros, 168
SmallChange, 151
sonido con visual basic, 366
sonido, 368
Sqrt, 140
SOUND, 369
StartSound, 367
Static, 39; 51
Step, 207
Stop, 260
Str$, 42; 69
StretchBlt, 355
Strings, 37
Style, 147
Sub Main, 365
Sub, 50
submenú, 97
suceso
    Change, 32; 152; 168
    Click, 28; 171
    DragDrop, 238; 248; 249
    DragOver, 239; 248
    GotFocus, 126
    KeyDown, 105
    KeyPress, 66; 116; 237
    KeyUp, 105
    LinkClose, 308; 309
    LinkError, 308; 309
    LinkExecute, 309
    LinkOpen, 307; 308
    Load, 29
    LostFocus, 126
    MouseDown, 219
    MouseMove, 219
    MouseUp, 219
    Paint, 203
    PathChange, 168
    Timer, 340
    Unload, 339
suceso KeyDown y KeyUp, 67
suceso [illegible] 203

T

Tab, 211; 255
TabIndex, 255; 348
TabStop, 72
Tag, 236
tamaño de los controles, 23
tecla pulsada, 66
telFono, 295
temporizador, 22; 114; 115
Text, 68; 176
TextHeight, 200
texto, gráficos, 199
TextWidth, 200
Time$, 116
Timer, 340
tipo
    Control como argumento, 53
    Control, 241
    Currency, 38
    de un objeto, 241
tipos
    de datos en las llamadas, 328
    de datos Form y Control, 52
    de datos, 37
    de Windows, 333
Top, 193
trazas, 151
Type ... End Type, 58
Type, 123
TypeOf, 53; 241

U

UCase$, 66
Unload, 81; 111; 128; 339
UpdateWindow, 382
utilidades, panel, 21

V

Val, 42
valor
    de una propiedad, 25
    argumento por, 51
    ByVal, 52
Value (fac.), 75
Val..., 133; 135; 150
variable, 36
    declaración a nivel de la forma, 39
    declaración a nivel del módulo, 39
    estática, 51
    global, 39
    local, 39

    nombre, 36
variables con el mismo nombre, 40
ventana, 16
    de código, 28
    inmediata, 10; 32; 261
Visible, 81; 99
Visual Basic
    arrancar, 14
    instalar, 13
visualizar
    el código, 7
    una forma, 7; 110, 143

W

While, 46
Width y Height, 82
Width, 193; 197; 413
WIN.INI, 74
WM_CLEAR, 343
WM_COPY, 343
WM_CUT, 343
WM_PASTE, 343
WM_USER, 343
WriteComm, 377

X

Xor, 209; 235

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

PROGRAMACION DE APLICACIONES PARA AMBIENTE WINDOWS (VISUAL BASIC)

MATERIAL COMPLEMENTARIO

PRACTICAS

NOVIEMBRE - DICIEMBRE

1994

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

Lab 1: Creating an About Box

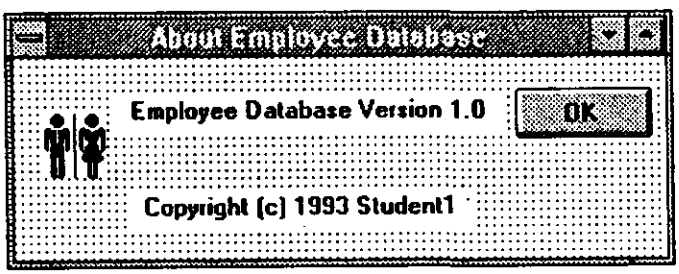
Objective

The overall objective for this lab is for you to become comfortable with the way the Microsoft® Visual Basic™ programming system for Windows™ supports application development. In this lab you will design and build an About box—that's the copyright form—for the Employee Database application that you will be building during class.

At the end of this lab, you will be able to create a single-form Visual Basic application with:

- Two label controls
- One command button control
- One picture box control

When you are finished, the form should look like this.



Procedures

Σ To get started

1. Start Visual Basic.
2. Rename and save the form as ABOUT.FRM in \STUDENT1\ABOUT.

Σ To create controls and set properties

1. Set the properties for the About box form as follows.

Caption	BorderStyle	Name
About Employee Database	1 - Fixed Single	frmAbout

2. Resize the form to make it smaller and narrower. For exact dimensions, see Appendix A, "The Employee Database Application Object List," in the back of this manual.
3. Set properties for each of the controls listed below.
4. To create the top label, double-click the label control tool in the Toolbox, and position the new control at the top of the form.
5. Set the properties for the top label as follows.

Control	Name	Caption	Alignment
Top label	lblTop	Employee Database version 2.0	2 - Center

6. Grab the sizing handles and lengthen the label so that the caption fits on one line. Center the label and move up.
7. To create the bottom label, double-click the label control tool in the Toolbox, again. Position this second label in the middle of the form.
8. Set the properties for the bottom label as follows and center it on the form.

Control	Name	Caption	Alignment
Bottom label	lblBottom	Copyright (c) 1992 Student 1	2 - Center

9. Grab the sizing handles and lengthen the label so that the caption fits on one line. Center the label and move up.
10. To add the command button in the upper-right corner of the form, double-click the button control tool in the Toolbox. Position the button in the upper-right corner of the form.
11. Set the properties for the button as follows.

Control	Name	Caption
Command button	cmdOK	OK

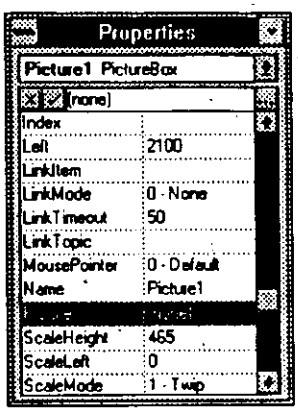
12. To add the picture box, double-click the picture box tool in the Toolbox, and position the picture box in the upper-left corner of the form.

13. Set the properties for the picture as follows:

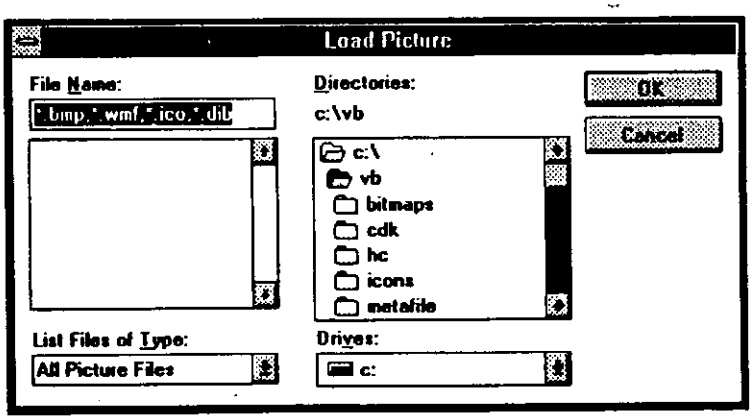
Control	Name	AutoSize	BorderStyle
Picture box	picLogo	True	0 - None

Σ To add a picture

1. To add the correct icon for the picture control, select the picture property, and then click the three dots(...) that are displayed at the right end of the settings box. It should look like this.



2. This will display a Load Picture dialog box.



3. In the \VB subdirectory, locate the \ICONS subdirectory and then the \MISC subdirectory. Select the MISC28.ICO file and choose OK. This will display the two figures as an icon on the form.

Σ To position the form for startup

Now that you have placed the controls on the form and set the properties for those controls, you might want to set the position for the form on the screen when it is started. This is a lot simpler than you might think.

- Place the form where you want it—for example, in the middle of the screen. When you save the form, it will retain the information about its position on screen.

Σ To save your work

Now you need to make sure that you save the changes you have made to the About box form. This process has two steps: saving the form itself and saving the project.

1. To save the form, choose Save File As from the File menu . Make sure that you save the form in the \ABOUT subdirectory. This will probably not be the default directory in the Save File As dialog box.
2. Choose OK.

   Next you need to save the project itself—in this case, only the About box form, but in other cases an entire list of forms.
3. From the File menu, choose Save Project As. You will get a dialog box that asks you to give the project a name.
4. Type **ABOUTBOX.MAK** in the form, and make sure that you are saving the file to the \STUDENT1\ABOUT subdirectory.
5. Choose OK.

If There's Time

You have completed the essential steps needed to get a simple About box up and running. In order to become familiar with some of the other properties of labels that you can set, alter the FontName, FontSize, FontBold, and FontItalic properties to further customize the look of the form.

Lab 2: Employee Database Application Specification

Objectives

There are two objectives for this lab: The first is to outline the general characteristics for a small front-end for an employee database; the second is to give you a chance to take a first pass at designing the application based on your current skill level.

Procedures

Imagine that you are sitting at your desk and you have just received a memo like this.

Memo

To: Joe Prototyper
From: High Management
Date: Today
Re: Front-end of our new employee database

We have just decided that we need a new front-end to our employee database and have heard the Visual Basic is just the product to do it with.

Please develop a working prototype for this application and make sure that it contains these functions:

1. Display a list of employees and their departments.
2. Display the following information about any employee selected from the list mentioned above:
   a. Employee name (last name, first name, MI)
   b. Employment status - full or part or temporary
   c. Department code
   d. Email alias
   e. Picture
   f. Payroll deductions
      Employment Stock Purchase Program
      401K
      United Way
3. Add new employees to the database.
4. When new employees are added, this application should be able to add a digitized version of a current photograph of the employee to the employee information display.
5. Allow administrators to print out detailed data on individual employees.

This application will also have a legal copyright notice.

The application should be very easy to use and be consistent with user interface guidelines provided by Microsoft.

Now You Do It

Based on the specification and your current understanding of user interface design guidelines, take a first pass at designing the interface for the class application.

Directions

You should probably work in small groups, but that is not required.

1. Decide how many forms your application might have.
2. Sketch in the menu items for the application.
3. Locate the controls on the forms.

Let's See What We Did

Now that you have had a chance to work on the user interface design for the class application, take a look at one possible way of solving the problem.

Walk Through — Introduction to the Employee Database Application

1. Start Employee Database from the Walk Throughs program group.

   This is the base form for the application. It contains two list boxes. One displays employee names and the other displays the employees' departments.

2. Click the File menu to show Exit.
3. Click the Help menu to show About.

   There are two menus, containing one item each. Choosing the Exit command from the File menu allows the user to exit from the application. Choosing the About command from the Help menu leads to an About box that displays copyright information about the application.

4. From the Help menu, choose About.

   The About box is a standard element in most commercially available applications. It normally appears as an item on the Help menu, and it is the legal statement of copyright ownership.

5. Choose OK.

   Close the About box and return to the Employee Database form.

6. Click the Peter Krauss name in the list box.
7. Choose the View button.

   Clicking this button lets the user see the employee information for the name selected in the Employee list box.

   This view of the Employee Record form will only display the contents of the database; it will not let the user change that information.

8. Choose the Done button.

   This returns you to the Employee Database form.

   Let's say that you are the database administrator and you need to add a new employee, one Claude Thibeau.

9. Choose the Add button.

   Clicking this button takes you to the Employee Record form where you add employees to the database.

   Here are the particulars on the new employee's employment status. Add this information in the correct controls.

Name:	Thibeau, Claude P.
Position Type:	Full Time
Department:	SAL
Email Name:	claudet
Deductions:	None

   The bitmap of his employee identification photo has already been added to the system, so you will need to find it to display it with the data.

10. Choose the Add Photo button.

    With this command, you can locate and display a current photo of the employee.

11. Select CPT.BMP.

    This file is located in \PHOTOS.

12. Choose OK.

    This adds the photo to the form.

13. From the Employee Record form choose Done.

    You are prompted to save changes. Choose OK to save changes.

    This returns you to the Employee Database form.

14. Select the employee Maria Lopez and choose the Update button.

    This button displays the Employee Record form and allows you to update the information.

15. Change the last name from "Lopez" to "AnotherName".

16. Choose Update.

17. From the Employee Record form choose Done.

    This returns you to the Employee Database form.

18. From the Employee Database form select Peter Krauss and choose the Delete button.

    You are prompted if you are sure you want to delete the employee.

    Choose OK to accept the delete.

    The employee is deleted from the list.

19. Choose the Exit button.

    This closes the application. Notice that the user can exit either from a command button on the form or from the menu. This approach is consistent with user interface guidelines.

Introduction to the Employee Database Application - Print Preview

If you are interested in the Print Preview function, start Visual Basic and open the project \SOLUTION\PREVIEW\EMPLOYEE.MAK.

1. From the Run menu choose Start.
2. From the Employee Database form, choose Peter Krauss.
3. Choose View.

   To preview a printout of an individual employee record, the user would...

4. Choose Print Preview.

   This displays the employee record details on a form that appears in the center of your display.

   By the way, this Print Preview functionality is an optional lab at the end of the course.

5. Choose Preview Cancel on the Employee Database form. (Notice that it is not on the Print Preview form.)
6. On the Employee Record form, choose Done.

   This returns you to Employee Database.

You have just taken the user's tour of the application, but you are all programmers. What does the application look like from the programmer's perspective?

The easiest way to look at the Employee Database application from the programmer's point of view is to look at Appendix A, "Employee Database Application Object List."

Open your student workbook. Take a detailed look at the forms and controls that make up the application.

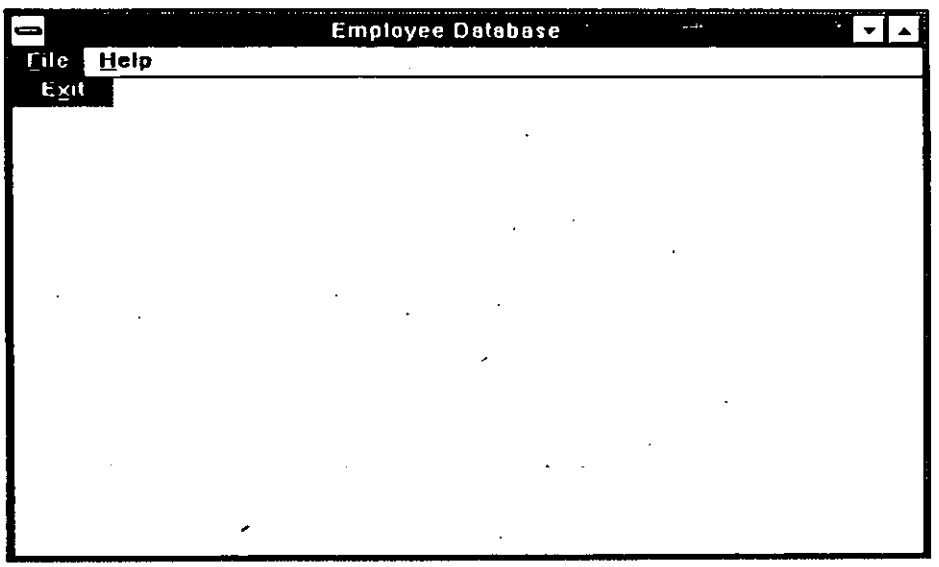
Lab 3: Creating Application Menus

Objective

You have the main forms for your database front-end. It is now time for you to add the menus.

At the end of this lab you will be able to create menus for single-form and multiform applications developed in Visual Basic.

Your output from this lab should look something like this.



Procedures

1. From the STUDENT1 program group, double-click the Menus icon.

   This starts Visual Basic with \STUDENT1\MENUS as your current working directory.

2. From the File menu, choose Open Project.
3. Double-click EMPLOYEE.MAK.
4. From the Project window, select EMPDB.FRM.
5. Choose the View Form button.

   This displays the lab version of the Employee Database form.

6. From the Window menu, choose Menu Design Window.
7. To create the File menu and its Exit item, set their properties as follows.

Control	Caption	Name	Indentation
File menu item	&File	mnuFile	0
Exit item	E&xit	mnuFileExit	1

Note Use the right arrow icon in the Menu Design Window to set the indentation to 1.

8. From the Menu Design Window, choose Next.
9. To create the Help menu and its About item, set their properties as follows.

Control	Caption	Name	Indentation
Help menu item	&Help	mnuHelp	0
About	&About...	mnuHelpAbout	1

10. From the Menu Design Window, choose OK.
11. Test to see that your menus work by clicking each of them.
12. From the File menu, choose Save Project.

    Save the project as EMPLOYEE.MAK in \STUDENT1\MENUS.

Lab 4: Loading and Unloading Forms

Objectives

So far, you have created the base forms and menus for the Employee Database application. Now it is time to connect those forms so that they behave a little more like a Windows-based application.

At the end of this lab, you will be able to code an event procedure so that it:

- Makes a form invisible.
- Displays one form and hides another.
- Unloads a form from memory.
- Closes a Visual Basic application.

At the end of this lab, you will have made the following changes to these files.

File	Changes	Declarations/Procedures
EMPDB.FRM	Display the About form.	mnuHelpAbout_Click
	Close the application.	mnuFileExit_Click
	Display the Employee Record form and hide the Employee Database form.	cmdView_Click
ABOUT.FRM	Unload the About form.	cmdOK_Click
EMPREC.FRM	Hide the Employee Record form and display the Employee DB form.	cmdDone_Click

Before You Begin

Before you start this lab, get out a blank piece of paper. Use the paper as a mask to cover the page below the step where you are immediately working.

This lab is designed to be interactive. During it, you are asked questions that are designed to test your understanding of the concepts presented during the lecture. Usually the answer to the question is on the following lines, and the paper will cover the answer until you've had a chance to challenge yourself.

Exercise: Connecting the Forms of Your Application

High-Level Procedures

The following instructions give a high-level view of the steps required to complete this lab. More detailed instructions follow.

These are the five major phases required for successful completion of this lab:

1. Add the code that displays the About box form from the Employee Database form when the user chooses the About command from the Help menu.
2. Add the code that closes the Employee Database application from the Employee Database form when the user chooses the Exit command from the File menu.
3. Add the code that hides the Employee Database form and displays the Employee Record form when the user chooses the View command button on the Employee Database form.
4. Add the code that hides the Employee Record form and displays the Employee Database form when the user chooses the Done command button on the Employee Record form.
5. Add the code that unloads the About box form when the user chooses the OK command button on the About box form.

   In each instance where you need to add code, there is a comment—remember, Visual Basic comments start with an apostrophe (')—that begins: `'*** Add here...` So all you have to do is choose the Find command from the Code menu to search for places where the string *** appears.

Detailed Procedures

To add the code that displays the About box form

1. From the Student1 program group, double-click the Connecting Forms icon.

   This starts Visual Basic with \STUDENT1\CONNECT as your current working directory.
2. From the File menu, choose Open Project.
3. Double-click EMPLOYEE.MAK.

   Make sure you are in the \STUDENT1\CONNECT subdirectory.
4. Stop here! Which form do you need to open in order to find the event procedure that invokes the About box?

   _____

   If you said the Employee Database form, you were correct.
5. Select the appropriate form name.

   Note We have added a View button to this form. This is provided because you will need some kind of mechanism for moving from one form within the application to another. You will also find a Done button on the Employee Record form.
6. From the Project window, choose the View Code button.

7. In the Object list box, scroll to the mnuHelpAbout object. Then in the Procedure list box, scroll to the Click event procedure.

   Now you have come to the heart of the matter. You need to insert a statement that names the appropriate Visual Basic method to display the About form.

   What is the syntax for that?

If you said frmAbout.Show, you were correct. Or, if you wanted to make the About box form modal and you said frmAbout.Show 1, you were also correct.

8. Add either of the preceding statements to the event procedure.
9. Keep the Code window open.

∑ To add the code that closes the Employee Database application

Now you need to add the code that closes the application when the user chooses the Exit command from the File menu.

1. What event procedure do you need to go to next?

If you said mnuFileExit_Click, you were correct.

2. Use the list box at the top of the EMPDB.FRM code window to locate that event procedure. Select the procedure name.

   What statement do you need to use to close the application completely?

If you said End you were correct.

3. Add this statement to the mnuFileExit_Click procedure.
4. Keep the Code window open.

∑ To add the code that hides the Employee Database form and displays the Employee Record form

You should be experienced at this by now.

- What event procedure do you need to go to in order to implement this?

Tip This uses one of the command buttons that we gave you.

If you said, without hesitation, cmdView_Click, then you are indeed experienced.

What is the syntax for showing the Employee Record form?

If you said frmEmpRec.Show, then you were right.

What is the syntax for hiding the Employee Database form?

If you said frmEmpDB.Hide, then you were right.

Note As a test of your understanding, the detailed procedures for the next set of changes have been omitted. You should be able to make the changes to these forms using the methods outlined above. Complete phases described below.

Σ **Add the code that hides the Employee Record form and displays the Employee Database form**

Σ **To add the code that unloads the About form and to set the Startup Form**

When you finish, test your application.

1. From the Run menu, choose Start.

   But before you go much further, you should consider this: You have three forms out there and only one of them will be displayed at startup. Which form do you think that should be?

If you said frmEmpDB, you were correct.

The next question is this: How do you set the startup form?

---

2. From the Options menu of Visual Basic, choose Project.

   This reveals a dialog box containing a number of selections.

3. Select the Start Up Form option.

   This places the option in the drop-down list box.

4. Using the cursor, select the drop-down arrow on the list box.
5. Select EmpDB to be the startup form.
6. Choose OK.

   Now you can continue testing work.

7. Choose the View button. This displays the Employee Record form.
8. Choose the Done button. This hides Employee Record and displays Employee Database.
9. From the Help menu, select About. This displays the About form.
10. On the About form, choose OK. This unloads the About form.
11. From the File menu, choose Exit. This unloads the entire application and returns to the Microsoft Visual Basic screen in design mode.

Σ **Save your work**

Lab 5: Adding Controls to Forms

Overview

The overall objective of this module is for you to finish creating the user interface for the Employee Database that you are building. This will include completing the following forms:

- The Employee Database form

  This is the startup form for the application. It allows the user to select employee information by department and will bring up the Employee Record form.

- The Employee Record form

  This form allows the user to enter personnel data about the individual into the Employee Database. Starting from this form, users will be able to locate a picture file on the employee.

At the end of this lab you will have the forms and controls necessary for the modified implementation of the application up to the printing lab. In later labs, you will add any code needed to make the application work.

Procedures

Some controls and properties have been provided for you in this lab's project files. You will have to create additional controls and set additional properties as outlined.

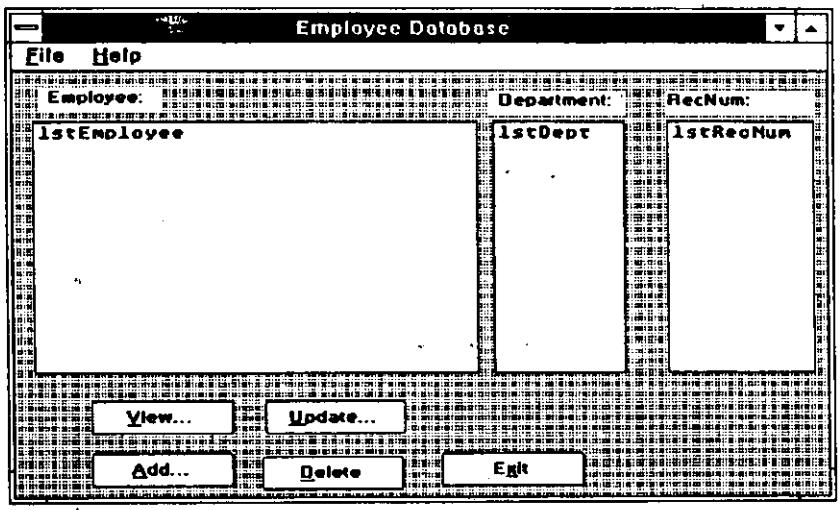
Exercise: Completing the Employee Database Form

1. From the STUDENT1 program group, double-click the Adding Controls icon.

   This starts Visual Basic with \STUDENT1\CONTROLS as your current working directory.

2. From the File menu, choose Open Project.
3. Double-click EMPLOYEE.MAK.
4. From the Project window, select EMPDB.FRM.
5. Choose View Form.
6. Notice that one list box and its label have already been provided for you on the right side of the form. This list keeps track of information the application needs but the user does not. You will resize the form before you save it to hide this list.

   When you finish working on this form, it will look like this.



You will add:

- The Employee label and list box
- The Department label and list box
- The Add and Exit buttons

Note For exact information on position and size for each of these controls, see Appendix A, "Employee Database Application Object List." If you want, feel free to approximate the placement.

---

7. Add the two labels for the Employee and Department list boxes by double-clicking the label tool in the Toolbox twice. Position them in the upper-left corner and upper center of the form. Set their properties as follows.

Control	Name	Caption	Alignment
Left list box label	lblEmployee	Employee:	0 - Left Justify
Middle list box label	lblDept	Department:	0 - Left Justify

8. Add the two list boxes for the Employee and the Department list boxes by double-clicking the list box tool in the Toolbox twice. Position the two list boxes on the form and set their properties as follows.

Control	Name	FontName	FontSize
Left list box label	lstEmployee	FixedSys	9
Middle list box label	lstDept	FixedSys	9

Note The first character of each of the preceding control names is a lower case "L". Visual Basic control names must start with a letter, not a number. Also, be sure to change the FontName property to FixedSys. If you do not, the font sizes available to you will not be correct.

9. Using the sizing border, resize the form by moving the right edge inward so that lstRecNum is no longer visible.
10. Add two additional command buttons for manipulating employee information and canceling the form by double-clicking the command button tool in the Toolbox twice. Set the properties for these buttons as follows.

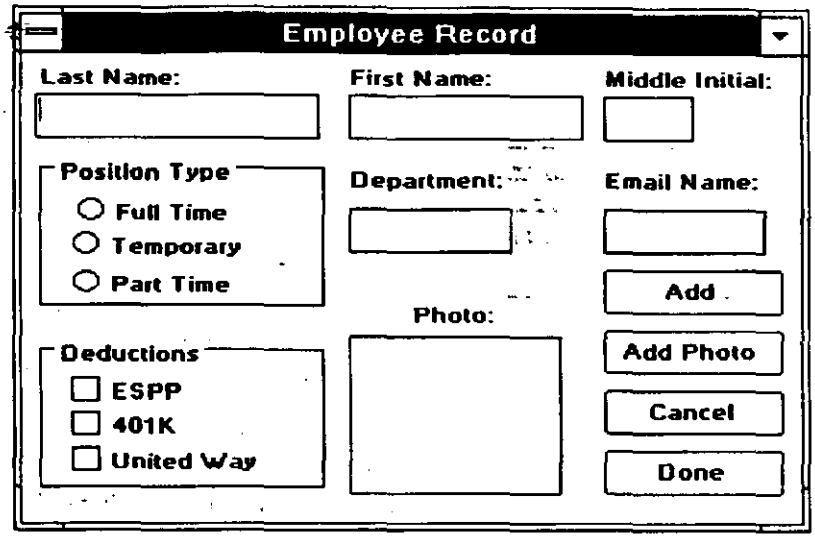
Control	Caption	Name
Add button (below View)	&Add...	cmdAdd
Exit button (to the right of Delete)	E&xit	cmdExit

This completes placing controls on the Employee Database form.

11. From the Control menu on the Employee Database form, choose Close.
12. From the Project window, select EMPREC.FRM.
13. Choose View Form to show the starting version of this form on screen.

Exercise: Completing the Employee Record Form

The final version of this form will look like this.



You will add:

- The Deductions frame and check boxes
- The Department label and text box
- The Email Name label and text box
- The Photo label and picture box
- The Add Photo and Cancel buttons

Note The lblListNum control provided in the center of the form is a hidden control that is used for record-keeping purposes. It will not appear on the form at run time, and code for using it will be supplied in a later lab.

1. Add an additional frame control for the Deductions group by double-clicking the frame tool in the Toolbox. Position it on the left side of the form under the Position Type frame.

   Warning In order for frames to retain the controls you set on them, you must first place the frame and then draw the controls on (not double-click) them.

2. Set the properties of the Deductions frame as follows.

Control	Caption	Name
Deductions frame	Deductions	fraDeductions

3. Double-click the check box tool and draw the check box inside the frame. Repeat these steps for each of the three check boxes.

Control	Caption	Name
ESPP check box	ESPP	chkESPP
401K check box	401K	chk401K
United Way check box	United Way	chkUWay

4. Add the two labels that are used for the Department and Email Name controls by double-clicking the label tool twice. Position these two labels in the middle of the form and to the right of the form, respectively. Set the properties for the labels as follows.

Control	Caption	Name
Department label	Department:	lblDeptName
Email Name label	Email Name:	lblEmailName

5. Add the two text boxes that are used for the Department and Email Name controls by double-clicking the text box tool in the Toolbox twice. Position them below the two labels, and then set the properties of the text boxes as follows.

Control	Text	Name
Department text box	(blank)	txtDeptName
Email Name text box	(blank)	txtEmailName

6. Add the label over the Photo box by double-clicking the label tool in the Toolbox. Position this label below lblListNUm. Set the properties for the label as follows.

Control	Caption	Name
Photo label	Photo:	lblPhoto

7. Add the picture box by double clicking the picture tool in the Toolbox. Position it in the lower center of the form. Set the properties for the picture as follows.

Control	Name
Picture box	picEmployee

8. Finally, add two additional command buttons to the right side of the form by double-clicking the command button tool in the Toolbox twice. Position the command buttons, and then set their properties as follows.

Control	Caption	Name
Add Photo button (below Add)	Add &Photo	cmdAddPhoto
Cancel button (above Done)	&Cancel	cmdCancel

You are now finished building the two major forms in this application. You will be building two additional smaller forms in later modules.

9. From the Control menu on the Employee Record form, choose Close.

Now save all of your forms and the project.

10. From the File menu, choose Save Project. You should already be in \STUDENT1\CONTROLS.
11. Save the project as EMPLOYEE.MAK.

Lab 6: Using Constants and Variables

Objectives

At the end of this lab you will be able to:

- Define global constants in a .BAS module
- Define global variables in a .BAS module
- Declare form-level variables in a .FRM file

At the end of this lab, you will have made the following changes to these files.

File	Changes	Declarations/Procedures
EMPLOYEE.BAS	Global constant declarations	
	Global variable declarations	
EMPDB.FRM	Local variable declaration	Form_Load
	Local variable declarations	cmdView_Click
EMPREC.FRM	Form-level variable declarations	General Declarations

Before You Begin

Notice that we have provided a final form of the browser for this and following labs.

Exercise: Adding Data Declarations to Your Application

High-Level Procedures

The following instructions give a high-level view of the steps required to complete this lab. More detailed instructions follow.

There are five major phases required for successful completion of this lab. For specific details on constant and variable names and their initial values, see the appropriate tables in the Detailed Procedures section that follows.

1. Copy already-coded variables and constants into the appropriate file.
2. Add the appropriate global constant declarations to a module.
3. Add the appropriate global variable declarations to a module.
4. Add one form-level integer variable and one string variable (called Position and Deductions, respectively). Use the position variable to hold information about an employee's position type (Full Time, Temporary, or Part Time). Use the Deductions string as a three-character string to store information about an employee's deductions.
5. Add a declaration for a local integer variable (called RecordNumber) that will be used to track the location in an array for a specific employee's data, and add a declaration for a local integer variable (called ListEntry) to be used to keep track of the ListIndex number for a specific employee in the lists on the Employee Database form.

Detailed Procedures

∑ To copy already coded variables and constants into the appropriate file

1. From the STUDENT1 program group, double-click the Constants and Variables icon.

   This starts Visual Basic with \STUDENT1\DATA as the current working directory.
2. From the File menu, choose Open Project.
3. Double-click EMPLOYEE.MAK.

   Make sure you are in the \STUDENT1\DATA subdirectory.
4. In the Project window for EMPLOYEE.MAK, click EMPLOYEE.BAS.
5. Select View Code.

   An empty Code window with the caption EMPLOYEE.BAS will appear in the center of your screen.
6. From the File menu, choose Load Text.
7. Locate and merge EMPLOYEE.TXT.

Note Use the Merge command button here to add code at the bottom of the file, not the Replace command button.

Σ To add the appropriate global constant declarations to the .BAS module

Having found and loaded the .BAS module, you are now ready to begin adding a number of constant declarations to this file.

Remember, in each instance where you need to add code, there is a comment that begins: `'*** Add here...`. So all you have to do is choose Find from the Code menu to search for places where the string `***` appears.

- Add each of the following constants along with their initial values to EMPLOYEE.BAS.

Constant	Value
UNCHECKED	0
CHECKED	1
OK_AND_CANCEL	1
OKAY	1
CANCEL	2

Note Did you remember that you need to specify to Visual Basic what kind of constants these are?

Σ To add the appropriate global variable declarations to the .BAS module

Having added the global constants to EMPLOYEE.BAS, you are now ready to add a couple of variable declarations to the same file.

Remember here, too, that you need to specify to your application what kind of variables these are.

Scroll to just below the user-defined data type EmpRec at the bottom of EMPLOYEE.BAS. This portion of the file deals with a number of variables that the application needs in order to manage the list of employees and their data.

Variable name	Data type	Comment
PositionType	String	Variable-length string
NextLocation	Integer	Stores next unused location in the array

Σ To add one form-level integer variable and one fixed-length string variable to the employee record form

So far in this lab you have worked with global-level declarations. Now you are going to do some form-level declarations.

1. If you haven't saved and closed EMPLOYEE.BAS, do so now.
2. Select EMPREC.FRM in the EMPLOYEE.MAK Project window.
3. Select View Code.
4. In the Object list box, select General. In the Procedures list box, select Declarations.

   At this point, you are going to be declaring the following.

Item	Data type	Comment
Position	**Integer**	
Deductions	**String**	Length represented by the constant MAXDEDUCTIONS

Σ To add declarations for two local integer variables to the Employee Database form

So far, you have been working with global- and form-level declarations, but as the presentation on scoping indicated, you can also declare variables locally. For this phase and the next one, you will need to be in the code for the Employee Database form—EMPDB.FRM.

1. If you haven't already saved and closed EMPREC.FRM, do so now.
2. Select EMPDB.FRM in EMPLOYEE.MAK.
3. Select View Code.
4. The EMPDB.FRM code window should appear in the middle of your screen.
5. In the Object List box, click the down arrow.
6. Select the procedure for the View command button.
7. At the top of this event procedure, you need to declare a couple of list-management variables for later use. More will be said about that in a later lab. For the time being, declare these two variables and make sure that you indicate their type.

Name	Data type
RecordNumber	**Integer**
ListEntry	**Integer**

8. From the File menu, choose Save Project.

   Save the project as EMPLOYEE.MAK in \STUDENT1\DATA.

Lab 7: Writing Procedures

Objectives

At the end of this lab you will be able to:

- Define general procedures in a module file (a .BAS file).

At the end of this lab, you will have made the following changes to this file.

File	Changes	Declarations/Procedures
CLEARREC.BAS	General procedure definitions	ClearEmpRec ClearDeductions

Exercise: Adding Code to Your Application

High-Level Procedures

The following instructions give a high-level view of the steps required to complete this lab. More detailed instructions follow.

There are two major phases required for successful completion of this lab.

1. Add the appropriate statements to assign values to control properties to a general procedure in a .BAS module that clears all of the controls in the Employee Record Form.
2. Create a general procedure in a .BAS module to clear all the deduction check boxes on the Employee Record form.

   Remember, in each instance where you need to add code, there is a comment that begins: `'*** Add here...` So all you have to do is choose the Find command from the Code menu to search for places where the string `***` appears.

Detailed Procedures

Σ To add the appropriate statements to assign values to control properties in a general procedure

1. From the STUDENT1 program group, double-click the Writing Code icon.

   This starts Visual Basic with \STUDENT1\CODE as your current working directory.

2. From the File menu, choose Open Project.
3. Double-click EMPLOYEE.MAK.

   Make sure you are in the \STUDENT1\CODE subdirectory.

4. In the Project window for EMPLOYEE.MAK, click CLEARREC.BAS.
5. Select View Code.
6. You will be in the General Declarations section. Select ClearEmpRec from the procedures list.

   You will add code to flesh out some of the details in a general procedure that clears all of the controls in the Employee Record form when the user chooses either the Add button or the Cancel button on that form.

   In essence, what this procedure does is to set all of the properties for all of those controls to either blank or default settings. The three controls that you are interested in (with the properties of interest) are listed below:

   - Department text box—Text property
   - Full Time Position Type check box—Value property
   - Employee Photo picture box—Picture property

   There are details in the comments marked with *** regarding the values to assign to the properties.

Σ Create a general procedure in a .BAS module

1. If you look carefully at the ClearEmpRec general procedure, toward the bottom of it there is a call to another procedure called ClearDeductions. We have provided the skeleton of that procedure and left it to you to fill it in. You remember what the deductions were, don't you?
2. From the General Declarations section, select ClearDeductions from the Procedures list in the Code window.
3. Following is a list of the deductions and the properties that need to be changed.

Control Property	Value to assign
ESPP Value property	UNCHECKED
401K Value property	UNCHECKED
United Way Value property	UNCHECKED

4. From the File menu, choose Save Project.

   Save the project as EMPLOYEE.MAK in \STUDENT1\CODE.

Lab 8: Conditional Logic and Loops

Objectives

The overall objective for this lab is for you to become comfortable with Visual Basic implementations for writing code to control the flow of your program by using **If...Then...Else** and **Select Case** statements.

You will also learn how to use loops to manipulate items in a list. In this lab you will add code to several different procedures that need to test certain conditions to decide which action to take depending on user actions or variable values.

At the end of this lab, you will be able to:

- Code a **For** loop to go through an array of user-defined type variables, take elements from the current user-defined type variable, and place them into different lists, using the **AddItem** method.
- Use the ListIndex property of a list to make a given item in the list the default user selection.
- Code **If...Then...Else** statements.
- Code **Select Case** statements that test conditions depending on user interaction, control properties, or variable values and based on those conditions alter the flow of the program appropriately.

At the end of this lab, you will have made the following changes to the following files:

Form	Procedure	Code to add
EMPDB.FRM	Form_Load	**For** Loop
EMPDB.FRM	Form_Load	Set default list selection
EMPDB.FRM	Sub FillFields	**If...Then...Else** statement
EMPDB.FRM	Sub FillFields	**Case** Statement
EMPREC.FRM	cmdAddPrintUpdate_Click	**Case** Statement

Exercise: Using Conditional Logic and Loops

The following instructions give a high-level view of the steps required to complete this lab. More detailed instructions follow.

High-Level Procedures

There are five major phases required for successful completion of this lab:

1. Coding a **For** loop to add items to a list
2. Modifying the ListIndex property to set a default list selection
3. Coding an **If...Then...Else** statement
4. Filling a **Select Case** statement skeleton
5. Coding a **Select Case** statement skeleton

Detailed Procedures

1. From the STUDENT1 Program group, double-click the Conditional Logic and Loops icon.

   This starts Visual Basic with \STUDENT1\LOGIC as the current working directory.
2. From the File menu, choose Open Project.
3. Double-click EMPLOYEE.MAK.

   Make sure you are in the \STUDENT\LOGIC subdirectory.
4. In the Project window, click EMPDB.FRM.
5. Choose the View Code command.

Σ To code a For loop

Since the code in the **For** loop is what causes the employee data for the application to be loaded into the lists on EMPDB.FRM, you will code the **For** loop before the **If...Then...Else** and **Select Case** statements.

For this part of the lab, you will be coding a **For** loop to do the following:

- Get the last name, first name, and middle initial for an employee from the EmpData array and place it in the Employee list on the Employee Database form.
- Get that employee's department from the same array and place that in the Department list on the Employee Database form.
- Get the record number from the array and place that data in the hidden list (lstRecNum) on the Employee Database form.

In order to do this:

1. Locate the event procedure Form_Load.
2. To decide how many times the **For** loop must be executed, check the general procedure LoadEmpDetails to see how many employees are originally loaded into the array EmpData and what subscript range they use.

   What is that range?

If you said 0 to 2, you were correct.

3. Return to the Form_Load procedure to code the opening statement for the **For** loop with the appropriate range.
4. Within the Form_Load procedure, find the commented section that starts:

```
'*** Add here a For loop ....
```

5. Set your **For** loop counter (already declared in the Form_Load procedure as I) to use the range you found in LoadEmpDetails.
6. The first statement to be included inside the **For** loop has been included in the comments so you can see an example of the syntax required to access separate elements of user-defined types in an array.
7. You will need to add the next two statements, one to add an employee's department information to the list lstDept on frmEmpDB and another to add the employee's record number information to the list lstRecNum on frmEmpDB.
8. Don't forget to close the **For** loop.

Σ To make a given item in a list the default user selection

- Set the default selection to the top item in the list lstEmployee by setting the ListIndex property of this list to the appropriate value.

> **Tip** The appropriate value is in the comments in the lab code.

Σ To code an If...Then...Else statement

1. You should still be in EMPDB.FRM.
2. Go to the general procedure FillFields.
3. Find a section of comments that starts:

```
'*** Add here an If Then Else statement...
```

4. Add an **If ... Then ... Else** statement that will do the following:

   if PictureFile isn't an empty string,
   load into the picture box on the
   Employee Record form the filename
   contained in the PictureFile
   element of the current employee's record;
   otherwise,
   clear the picture box on the Employee
   Record form of any picture left
   there from a previous employee record.

   For more detailed information, check the comments in the code.

∑ To fill a Select Case statement skeleton

1. Stay in the general procedure FillFields in EMPDB.FRM for the next part of the lab.

   In this procedure you will find a partially complete **Select Case** statement that tests the value of the variable that stores the information regarding an employee's position type, using predefined constants for the cases. The code for the case for a Full Time employee is complete. It sets the appropriate property values for the Position Type option buttons and assigns a value to the string variable PositionType.

2. Add cases for Temporary and Part Time positions and all the code that will set the values for the appropriate option buttons and PositionType correctly in each case.

∑ To code a Select Case statement skeleton

1. Go to the event procedure cmdAddPrintUpdate_Click in the form EMPREC.FRM:

2. Add a "skeleton" **Select Case** statement to test the value of the caption on the cmdAddPrintUpdate command button, for the captions:

   "&Add"
   "&Update"
   "&Print"

   Check the comments in the code for the exact syntax you will need to use after the words "Select Case."

   For the time being, after each case, insert a **MsgBox** statement to report which case was selected. In later labs, code for two of the three buttons will be added to implement them more fully. In the extended solution, found in \SOLUTION\EXTENDED, the Update button is also implemented.

3. From the File menu, choose Save Project.

   Save the project as EMPLOYEE.MAK IN \STUDENT1\LOGIC.

Lab 9: Getting Output

Objectives

The overall objective for this lab is for you to become comfortable with writing code to create a report on an output form containing formatted text and, optionally, picture boxes with graphics and to send that form to the default printer under Windows.

At the end of this lab, you will be able to:

- Use the form properties CurrentX and CurrentY, various Font properties, the Cls and **Print** methods and the **Spc**, **Tab**, and string manipulation functions, such as **RTrim$** and **Mid$** to create formatted output on a form.
- Use the picture box property Image to copy a graphic from one picture box to another.
- Use **PrintForm** method to send this form to the default printer under Windows.

At the end of this lab, you will have made the following changes to this file.

Form	Procedure	Code to add
EMPREC.FRM	cmdAddPrintUpdate	The **Print** and **PrintForm** methods change some properties

Exercise: Getting Output

High-Level Procedures

The following instructions give a high-level view of the steps required to complete this lab. More detailed instructions follow.

There are two major phases required for successful completion of this lab:

1. Displaying formatted output to frmOutput.
2. Sending frmOutput to the default printer.

Before You Begin

Before you begin, make sure that you have used the control panel to set your default printer to go to a file so you will be able to test your work when you are done.

1. Double-click the Control Panel icon in the Main program group.
2. Double-click the Printers icon in the Control Panel program group.
3. Select the PostScript printer in the list of Installed Printers, and then press the Connect button.
4. Scroll down the Ports list in the Connect dialog box, until you find FILE. Click FILE and choose OK.
5. Choose the Set As The Default Printer button.
6. Then choose the Close button in the Printers dialog box.

Note You have been provided with the form that you need to complete this lab — OUTPUT.FRM. It is the form to which you will be printing employee records.

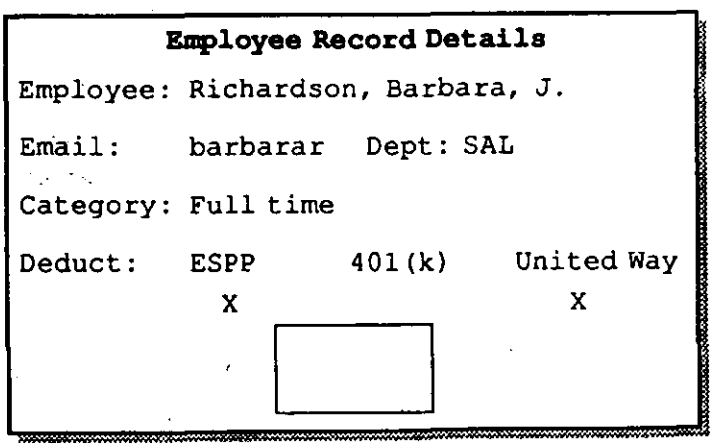
Detailed Procedures — Printing to Forms and Printers

∑ To display formatted output to frmOutput

1. From the STUDENT1 program group, double-click the Printing icon.

   This starts Visual Basic with \STUDENT1\PRINTING as the current working directory.
2. From the File menu, choose Open Project.
3. Double-click EMPLOYEE.MAK.
4. In the Project window, click the EMPREC.FRM.
5. Select the View Code option.
6. Locate the cmdAddPrintUpdate_Click event procedure.

   Before you start coding the application, you should probably review what the output is supposed to look like.



In the Case Print section of the **Select Case** statement that you created in a previous lab, add a series of **Print** statements that output individual records to a form called frmOutput. Do the following:

7. Immediately under the line Case &Print, add a statement to clear all previous output from the form frmOutput.
8. Next, tell the procedure which record to get for printing.

   Each control has a Tag property where you can store string data for any purpose that you like. The class application uses it to keep track of the position of the employee's personnel information in the array EmpData.

   To tell the procedure which record to get, you will need to complete two steps:

   a. Convert the Tag property for txtLastName from a string to a number, using the **Val** function.

   b. Then you need to assign that converted value to the integer variable RecordNumber.
9. Below the line that assigns a value to RecordNumber, add statements to set the CurrentX and CurrentY properties of form frmOutput to 0. This places the cursor in the upper-left corner of the form.
10. Next add statements to set the following values:

    FontName to "Courier"
    FontSize to 14
    FontBold to TRUE
11. Use the **Print** method to print the following string to the second print zone: "Employee Record Details" followed by a blank line. Remember that you use a comma to move to a print zone.

    If you look carefully at the form, you will notice that the size of the font has been changed and the bolding has been turned off for the record output information.
12. Now add statements to set the following values:

    FontSize to 12
    FontBold to FALSE
13. Then print out employee details listed in the commented area of the lab code, using the spacing indicated.
14. Next, load a copy of the graphic in picEmployee on form frmEmpRec into the picture box of the same name on the form frmOutput.

Σ To send frmOutput to the default printer

- Finally, send the form Output to the default Windows printer using the PrintForm method.

Exercise: Adding Print Preview Capability (Optional)

Your objective for this lab is to add a print preview capability to the application.

At the end of this lab, you will have made the following changes to these files.

Form	Procedure	Code To Add
EMPDB.FRM	cmdView	Change caption on cmdAddPhoto.
EMPREC.FRM	cmdAddPrintUpdate	Convert code that prints to frmOutput to a general procedure.
EMPREC.FRM	cmdAddPhoto	Add Print preview and Preview cancel functions.

As usual, check the code under \STUDENT1\PREVIEW for statements that start `'*** Add here...` to find the places to add the print preview code.

High-Level Procedures

You can use a strategy that you have already seen. Remember the top button on the Employee Record form? Its functionality and caption change depending on the use of the form—View, Add, or Update.

You can use that same strategy for adding new functionalities to the Add Photo button.

Detailed Procedures — Add a Print Preview Capability

Displaying the formatted output on frmOutput

At a more detailed level, your task during this optional lab is to implement the following functionalities:

- Adding a second capability to the Add Photo button (when the user has chosen View on the Employee Database form) called Print Preview.
- Adding a third capability to the Add Photo button (when the user has chosen Print Preview on the Employee Record form) called Preview Cancel.

Here are a few hints on how to implement print preview on the existing form:

1. First you should add code to the cmdView click event to change the caption of the Add Photo button on the Employee Record form to Print Preview.
2. Next you should convert the code that prints to frmOutput under the Case Print part of the cmdAddPrintUpdate click event code to a general procedure. This will let you call it both from the Print button and from the Print Preview button.

   To do this, move all of the code that you wrote for the Case Print in cmdAddPrintUpdate_Click (except for the last line that sends the output to the printer) into a new general procedure named PrintOutput.

3. Now place a call to this general procedure inside the Sub cmdAddPrintUpdate_Click to replace the code that you cut from the **Select Case** statement.

Note You will need to add the code to change the caption on the button to read "Print Preview". (In your code you will need an ampersand in front of the "v" so that it can be accessed through the keyboard.)

---

4. When the user chooses the View button on the Employee Database form, add the code to change the caption on the Add Photo button to make it a Print Preview button.
5. In the Click event for your enhanced Add Photo button, you will now need to add a **Select Case** statement to handle three possibilities for the value of the Add Photo button's caption:
   - Add &Photo, which will contain the original code for this button; appropriate when the user clicks the Add or Update button on the Employee Database form.
   - Print Pre&view, which will contain code to call the new general procedure PrintOutput, make frmOutput visible to the user and, lastly, change the caption on the button to "Preview Ca&ncel" to give user access to the third capability of this button.
   - Preview Ca&ncel, which will contain code to make frmOutput invisible and which will toggle the caption on the Add Photo button back to "Print Pre&view."

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

VISUAL BASIC

Material didáctico

Complemento

Diciembre 1994

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tel.. 521-40-20 Apdo Postal M-2285

Contents

Introduction

Introductions 3
Target Audience 4
Course Outline 5
Course Materials 8
Microsoft University Curricula 9
Facilities 10
Conventions 11

Module 1: Using Visual Basic

Overview 17
Visual Basic Tools 19
  Form and Project Windows 20
  The Toolbox and Properties Window 21
Building a Simple Visual Basic Application 24
Visual Basic Menu Commands 30
Summary 35
Lab Time 36

Module 2: Designing and Building Visual Basic Applications

Overview 39
Event-Driven vs. Procedural Applications 40
Visual Basic Terminology 42
Application Development 46
User Interface Design 47
User Interface Design Guidelines 51
Configuring Your Environment 53
Summary 55
Lab Time 56

Module 3: Working with Forms

Overview 59
Properties for Forms 61
  Message Boxes 63
Modal and Modeless Dialog Boxes 66
Multiple Document Interface Applications 70
Summary 73

Module 4: Laying Out Menus

Overview 77
Menu Guidelines 79
  Visual Basic Implementation 82
Summary 86
Lab Time 87

Module 5: Connecting Forms

Overview 91
Statements 93
Methods—Hide 95
Methods—Show 97
Event Procedures 101
Setting the Startup Form 103
Summary 104
Lab Time 105

Module 6: Using Controls

Overview 109
Types of Controls 111
Properties for Controls 112
Labels and Text Boxes 113
Masked Edit Control 116
Regular and 3-D Frames, Check Boxes, and Option Buttons 118
Regular and 3-D Command Buttons 123
Combo and List Boxes 125
Scroll Bars 132
Timers 135
Picture Boxes 138
Summary 141
Lab Time 142

Module 7: File Browsers and Other Controls

Overview 145
File Browsing 147
Scenario 148
Properties and Events for File Browser Controls 150
Change Events 153
The Change Event 154
PathChange 157
The Common Dialog Control 158
More Controls 159
Using the Grid Control to Display Output 160
3-D Panel 162
Group Push Buttons 165
Summary 169

Module 8: Using Visual Basic Data Types

Overview 173
Key Terms 174
Using Variables and Constants 175
Data Types of Variables 176
Constants 180

Scope 181
Scope of Data 182
Declaring and Using Local Variables 185
Declaring and Using Form-Level and Module-Level Variables 187
Declaring and Using Global Variables 189
Additional Visual Basic Data Types 191
Summary 195
Lab Time 196

Module 9: Writing Visual Basic Code

Overview 199
Visual Basic Procedures 201
Calling a Sub Procedure with Arguments 202
Types of Procedures 205
Creating the Event Procedure 206
General Procedure 208
Scope of General Procedures 209
Writing Code in Visual Basic 211
String and Numeric Conversion Functions 212
FormatS Function 214
Summary 216
Lab Time 217

Module 10: Using Conditional Logic and Loops

Overview 221
Control Structures 223
If...Then Blocks 224
If...Then...Else Blocks 225
Select Case Statements 227
Loops 230
Do...Loop 231
Do Until 235
For...Next 237
The GoTo Statement 240
Summary 241
Lab Time 243

Module 11: Debugging Code in Visual Basic

Overview 247
Debugging Terms 248
Debugging Code in Visual Basic 249
Summary 260

Module 12: Printing to Forms and Printers

Overview 263
Scenario 264
Methods for Printing 265
Print-Related Functions 268
Spc Function 269
Tab Function 270
Using PrintForm 271
Summary 273
Lab Time 274

Module 13: Data Access Using the Data Control

Overview 277
Overview of a Database 278
Overview of a Table 279
How Does Visual Basic Access Databases? 280
The Data Control 281
Binding Controls to the Data Control 283
Data Control Walkthrough 284
Data Control Methods and Properties 286
A Sample Application 287
Summary 288

Appendix A: Employee Database Application Object List 289

Appendix B: Employee Database Application Source Code 305

Introduction

Introductions

- Name
- Company Affiliation
- Title/Function
- Job Responsibility
- Programming Experience
- Expectations

Target Audience

Applications Developers, Prototypers, and Analysts Who Want to Know More About:

- The Essential Functions of the Visual Basic Product
- The Conventions of a Graphical User Interface
- The Essentials of the Basic Language

Course Outline

- **Using Visual Basic**

  Lab: Creating an About Box

- **Designing and Building Applications**

  Lab: Employee Database Application Specification

- **Working with Forms**
- **Laying Out Menus**

  Lab: Creating Application Menus

- **Connecting Forms**

  Lab: Loading and Unloading Forms

Course Outline

- **Using Controls**

  Lab: Adding Controls to Forms

- **File Browsers and Other Controls**
- **Using Visual Basic Data Types**

  Lab: Using Constants and Variables

- **Writing Visual Basic Code**

  Lab: Writing Procedures

Course Outline

- **Using Conditional Logic and Loops**

  Lab: Conditional Logic and Loops

- **Debugging Code in Visual Basic**
- **Printing to Forms and Printers**

  Lab: Getting Output

- **Using the Data Control**

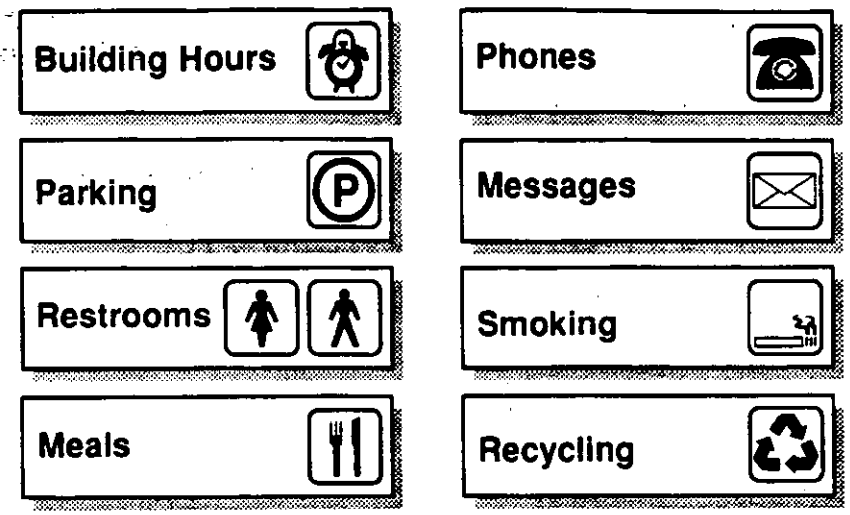
Course Materials

- Name Card
- Student Workbook
- Lab Manual
- Evaluation Form
- Reference Materials

Microsoft University Curricula

- Microsoft® MS-DOS® Operating System
- Programming In Microsoft Visual Basic™
- Microsoft Windows™
- Microsoft C and C++ Programming
- Microsoft Excel Programming
- Microsoft LAN Manager
- Microsoft Mail
- Microsoft SQL Server
- Management Education for Client-Server Computing

Facilities


Building Hours
Phones
Parking
Messages
Restrooms
Smoking
Meals
Recycling

Conventions

- Typographic and Naming Rules
- Program Groups and Lab Code Conventions

Sample Code

All code examples are written using monospaced Courier font, for example:

```
Sub Picture1_Click ()
   Picture1.Move Picture1.Left+750, Picture1.Top-550
End Sub
```

Visual Basic Terms

Microsoft Visual Basic programming terms are set in italic the first time they occur, for example:

Event procedures can be associated with a control or a form. They will be executed when a user or the system triggers the associated event.

Visual Basic Keywords

Words in bold are keywords used to write Visual Basic code, for example:

> Event procedures are always Sub procedures. When you define an event procedure, Visual Basic automatically codes the **Sub** and **End Sub** statements into the procedure.

You can reference Visual Basic terms and keywords through the online Help or the Visual Basic documentation.

For More Help

Each module contains pointers to the documents and specific chapters within the document that support the topics presented in the module. For additional information on using the Windows online Help facility, see Chapter 2 of the *Microsoft Windows User's Guide*.

Naming Convention for Controls

Visual Basic keeps track of all controls used in an application and lists them alphabetically. You will want to take advantage of this by adding prefixes to all the controls in your application. The following conventions have been used in the lab code. It is recommended that you adopt the following naming conventions in your own applications.

Control type	Name prefix
Check box	chk
Combo box	cbo
Command button	cmd
Directory list box	dir
Drive list box	drv
File list box	fil
Form	frm
Frame	fra
Grid	grd
Horizontal scroll bar	hsb
Image	img
Label	lbl
Line	lin
List box	lst
Menu	mnu
OLE client	ole
Option button	opt
Picture box	pic
Shape	shp
Text box	txt
Timer	tmr
Vertical scroll bar	vsb

Note You will need to establish your own names for the additional controls available in the Professional Edition of Visual Basic.

Program Groups

You will be working directly with three Program Groups during this course. The Walk Throughs program group is located on the left side of the Program Manager and contains the finished code for each of the sample applications used in the student manual. Student1, located on the right side of Program Manager, is the starting point for each of the labs in the course. Finally, the Solutions group, which is iconized at startup, contains a finished version of each of the labs in the course.

Lab Conventions

First and foremost, each of the labs is self-contained. This means that success in later labs is not dependent upon having completed earlier labs. Secondly, locations for fixes in code are marked by three stars, (***). To locate them, open the appropriate item in the Student1 program group, and select the View Code option in the Project window. When the Code window opens, choose the Find command from the View menu, enter "***", select the "All Modules" option, and then press ENTER.

Module 1: Using Visual Basic

Overview

- Visual Basic Tools
- Building a Simple Visual Basic Application
- Visual Basic Menu Commands

Overview

The purpose of this module is to introduce you to the key elements of Visual Basic. This is not intended to be an exhaustive review of all functions and tools; rather, it is designed to get you up and running on the product. Later modules will flesh out all of the details that you need to know to become Visual Basic programmers.

The best way to learn about Visual Basic is to review the individual functions in each major portion of the Visual Basic interface and then walk through the steps you should follow to develop an application.

The module is divided into three major sections. The first major section is a lecture about the elements of the Visual Basic application that you use in the creation of applications. The second is a demonstration where the trainer walks you through the application design process and you end up with a compiled executable file. The final section is a lab that gives you hands-on practice using some of the tools that you have just been introduced to.

Prerequisites

There are two key skills that you need for success in this module:

- Practice in using a mouse
- Understanding of graphically based applications

Overall Objective

At the end of this module, you will have successfully created your first Visual Basic application.

Learning Objectives

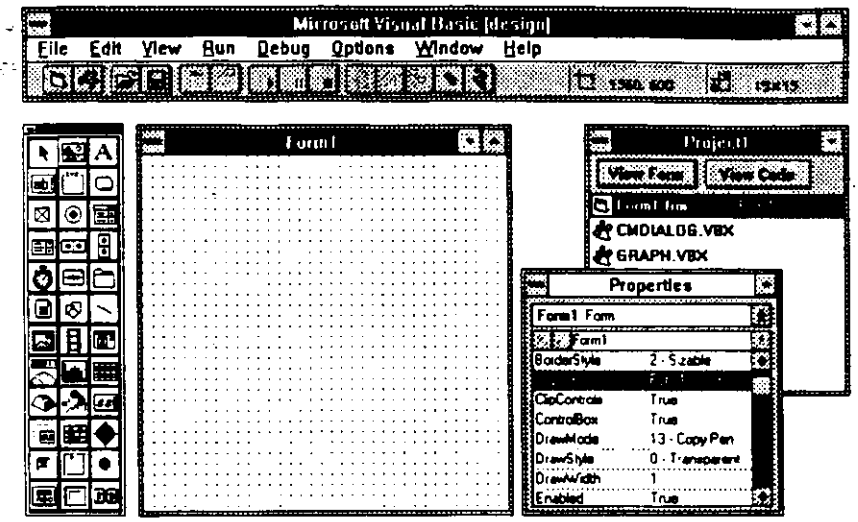
At the end of this module, you will be able to:

- Design an application user interface using the Form window, Toolbox, toolbar, and Property window.
- Name and save the application's forms and project files.
- Start and stop an application from the Visual Basic menu and/or the toolbar.
- Create an executable file and add it to a Windows group.

Σ Visual Basic Tools

- Form and Project Windows
- The Toolbox and Properties Window
- The Toolbar and Code Window

Form and Project Windows



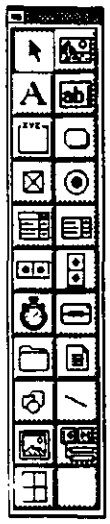
What Is a Form?

Forms are the heart of a graphically based application. They are what the user sees and interacts with in order to accomplish some task. They are also the place where you begin to build applications; on them you place controls—command buttons, list boxes, option buttons—that present the user with the choices that they have.

What Is the Project Window?

The Project window is a list that Visual Basic uses to keep track of the forms that you are using for your application. You will have as many .FRM files listed in the Project window as you have forms in your application. In addition you may have other files—.BAS and .VBX files—but we will hold off talking about those for a little while.

The Toolbox and Properties Window



What's in the Toolbox?

As the name suggests, the Toolbox is where you go to get the basic elements of every Windows-based application you create in Visual Basic.

There are two ways that you can place controls on a form, either by double-clicking the control tool in the Toolbox or by clicking and dragging the control. For the most part, either method has the same result. When you double-click a tool, the control shows up in the middle of the form; so you then have to drag the control to the correct position on the form.

Each control that you place on a form has a set of properties that you can use in order to get the right "look and feel" for your application. For the time being, all you need to know is that there are properties to each control. In later modules you will get a much closer look at the most commonly used properties for most of the tools in the Toolbox.

Custom Controls

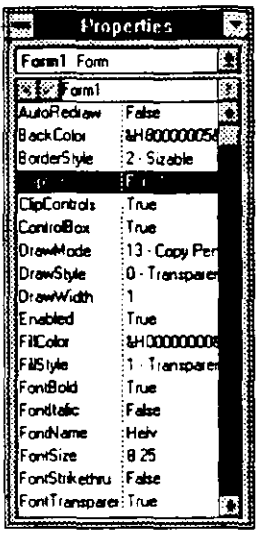
You can add controls to the Toolbox. Some third parties have created custom controls that you can purchase separately. The Professional edition of Visual Basic contains various custom controls.

Σ To Add a Custom Control

1. From the File menu, choose Add File.
2. Select the appropriate file. Custom Control files have a .VBX extension.
3. Choose OK.

What Are Properties and How Are They Set?

Properties for a control are set using the Properties window:

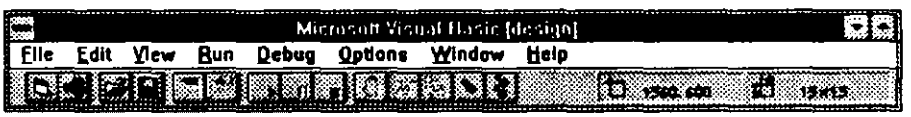


To set properties

1. Click the control whose properties you want set.
2. Scroll the Properties list on the Properties window until you find the property you want to set and select it.
3. Place the value for the property in the Settings combo box.
4. Click the check box to the left of the Settings combo box.

What Is the Toolbar?

The toolbar provides quick access to common commands or functions. These functions—like saving projects and starting the application during the design phase—are also available from the Visual Basic menus as well as through access and shortcut keys.



Nine of the relevant items on the toolbar are.

Toolbar	Menu path	Shortcut keys
New Form	From the File menu, choose New Form.	n/a
New Module	From the File menu, choose New Module.	n/a
Open Project	From the File menu, choose Open Project.	n/a
Save Project	From the File menu, choose Save Project.	n/a

Toolbar	Menu Path	Shortcut Keys
Menu Design window	From the Window menu, choose Menu Design.	CTRL + M
Properties window	From the Window menu, choose Properties.	F4
Start	From the Run menu, choose Run.	F5
Break	From the Run menu, choose Break.	CTRL + BREAK
End	From the Run menu, choose End.	n/a

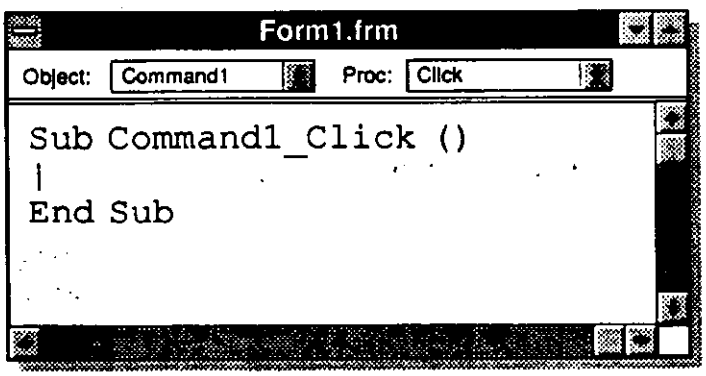
Note The rest of the functions deal with debugging and will be covered during that module.

Position and Size Coordinates

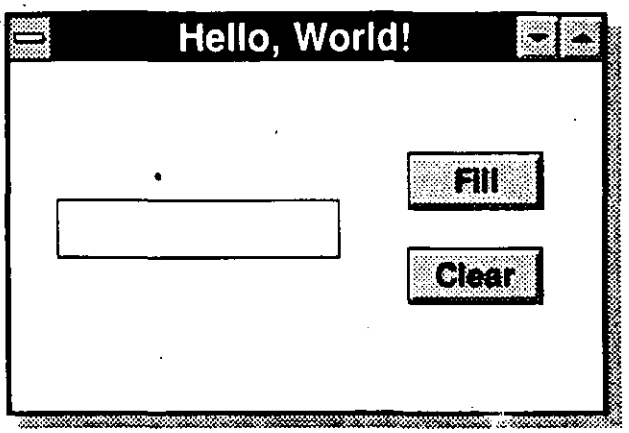
Visual Basic displays the coordinates for the upper-left corner for each control on the form relative to the inside upper-left corner of the form. It also displays the width and height of the control. You can set these values by placing the form or control approximately where you want it, or you can specify Left and Top coordinates from the Properties list. The unit of measurement is in twips (there are 1,440 twips in an inch). You will find out later that you can change this unit of measurement to something you are more familiar with.

What Is a Code Window?

Code windows display the code that implements your application. At first Code windows only contain the template for procedures and functions; you will add more code to them as you develop your application. There are several ways to open a Code window. The easiest way is to double-click the object whose code you want to view. For example, to locate the Click event template for a command button, double-click the command button on the form during design time.



Building a Simple Visual Basic Application



Walk Through — Building a Visual Basic Application

∑ To start WORLD

1. From the Walk Throughs program group, start WORLD.

   The purpose of this little application is to give you a chance to walk through most of the major steps needed for developing a Visual Basic application.

   How does the application work?

   It has one text box and two command buttons.

2. Choose the Fill button.

   When the user clicks the button, text is revealed in the text box.

3. Choose the Clear button.

   When the user clicks this button, the text in the text box is cleared.

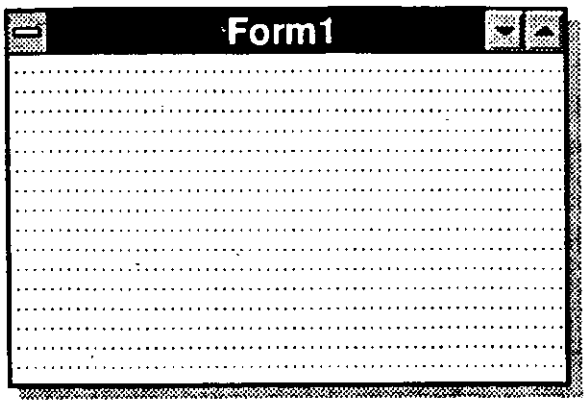
4. Quit the WORLD application.

   From the Control menu, choose Close.

Σ To start Visual Basic

1. Double-click the Visual Basic icon.

   Visual Basic will start with a blank form on the screen.



2. Resize the form to:

Height	2700
Width	4065

Σ To design the base form

The first step in designing the user interface is to set the properties for the application's base form. In this case, set the properties for the Hello, World! form.

1. In the Properties window Properties list, select BackColor.
2. Click the ellipsis (...) at the end of the text box.
3. Select a shade of green.

   The background of the form turns green. Selecting any color will automatically hide the color palette.
4. In the Properties list box, select Caption.
5. In the text box, type **Hello, World!**

   The Form title bar will contain Hello, World!
6. In the Properties window Properties list, select Name.
7. In the text box, type **frmWorld**

Σ To add controls and set properties

To create the form's controls, double-click the appropriate buttons in the Toolbox.

1. Double-click the command button tool in the Toolbox.
2. Set the following properties:

Caption	Fill
Name	cmdFill
FontSize	18
3. Move the Fill button to the upper-right corner of the form.

4. Double-click the command button tool in the Toolbox.
5. Set the following properties:

   Caption Clear

   Name cmdClear

   FontSize 18
6. Move the Clear button to the lower-right corner of the form.

7. Double-click the text box tool in the Toolbox.
8. Set the following properties:

   Name txtBox1

   Text Delete any text in the text box for this property.

Σ To add the Basic code to enable the controls

In this step, code will be added to activate the functionality of the controls in the form. The code will enable the text box to be filled with the words Hello World! when the user clicks the Fill button and enable the text box to be cleared when the user clicks the Clear button.

1. Double-click the Fill button that you just created to open the Code window.

   When you double-click this command button, Visual Basic brings you directly to the template for a procedure that responds to a Click event for this control. The names of event procedures follow the pattern:

   ```
   Sub objectname_eventname ()
   ```

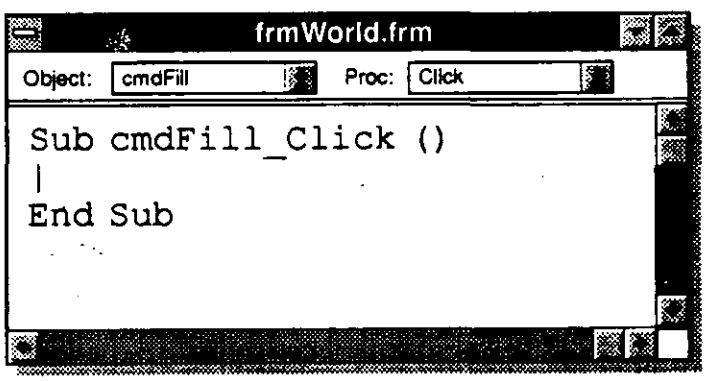
   In this case the procedure is named:

   ```
   Sub cmdFill_Click ()
   ```

   Also note that Visual Basic provides the statement that ends the subroutine:

   ```
   End Sub
   ```

   In the graphic below, notice that the object name shows up in the Object list box on the left, and the event name shows up in the Procedure list box on the right.



2. Add the following code:

```
txtBox1.Text = "Hello, World!"
```

Add a line of code between the two lines of the template for the Click event procedure. This code gives the application the target for the action (filling the text box) and the contents that you want assigned to the text box. In this case, "Hello, World!" is the contents.

3. In the Object list box, select cmdClear.
4. Add the following text:

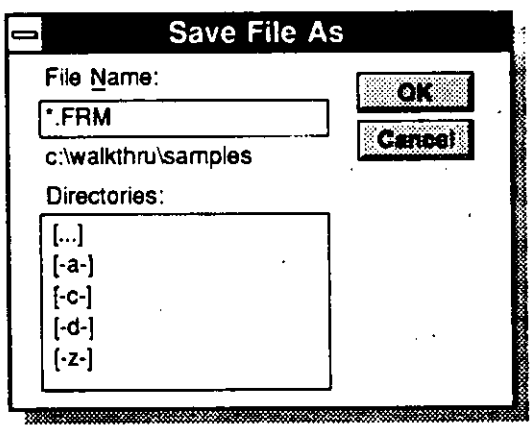
```
txtBox1.Text = ""
```

Inside the template for this Click event procedure, add a line of code to give the application the target for the action (clearing the text box) and the contents that you want assigned to the text box. In this case the contents is an empty string ("").

∑ To save your work

Before you create an executable version of this Visual Basic application that you can run directly in Windows, you should save the source code to disk. You will do this through two actions—saving the form and saving the project.

1. From the File menu, choose Save File As.



2. Make the current directory C:\WALKTHRU\SAMPLES.
3. Save the file as WORLD.FRM.
4. Choose OK.
5. From the File menu, choose Save Project As.
6. Save the file as WORLD.MAK.
7. Choose OK.

Σ To make an executable file

To create an executable file, make sure the source code for the project is open in Visual Basic.

1. From the File menu, choose Make EXE File.
2. Make the current directory C:\WALKTHRU\SAMPLES.
3. Save the file as WORLD.EXE.
4. Choose OK.
5. Minimize Visual Basic.

Σ To start the executable from Windows

You can start an executable file from Windows in two ways: 1) create a program group, and add the name of the file to the group; or 2) start Program Manager, open the File menu, and choose the Run command. We'll use the first method.

1. From the Program Manager File menu, select New.

   The New Program Object dialog box appears.
2. Select the Program Group option.
3. Choose OK.

   The Program Group Properties dialog box appears.

   In the Group Properties Description control, name the group something like SAMPLES.
4. In the Group File control, type **SAMPLES**
5. Choose OK.

   A blank window will appear.
6. From the Program Manager File menu, choose New.

   The New Program Object dialog box appears with the Program Item option selected.
7. Choose OK.

   The Program Items Properties dialog box appears.
8. Name the application.

   In the Description control, name the application WORLD or another name that reflects its function.

   Now comes the tricky part. You need to help Windows locate the executable file that you created.
9. Click the Browse button.

   The Browse window will appear.
10. Select drives and directories to locate WORLD.EXE.

    It should be in C:\WALKTHRU\SAMPLES.

11. Choose OK.

    The Program Items Properties dialog box appears with WORLD.EXE in the Command Line text box.

12. Choose OK.

    The SAMPLES Program Group will appear containing WORLD and a program icon.

∑ To run your application

- Double-click the World icon.

The Hello, World! application will start in the same screen location where the form was created.

∑ To stop your application

1. Click the Control menu in the upper-left corner of the Hello, World! form.

   A menu will appear.

2. Choose the Close command.

   Windows closes the application for you.

Visual Basic Menu Commands

- Managing Forms and Projects: The File Menu
- Editing Visual Basic Code: The Edit Menu
- Testing Applications During Development: The Run Menu
- Visual Basic Window Management: The Window Menu
- Getting More Information: The Help Menu

In order to develop the simplest of applications, there are a number of Visual Basic commands that you need to know about. Below is a brief listing of these commands. The list is not exhaustive; there are a couple of topics that have been left to a later module.

Managing Forms and Projects: The File Menu

- Adding a New Form
  File menu, New Form
- Adding an Existing Form to a Project
  File menu, Add File
- Deleting a Form from a Project
  File menu, Remove File
- Making an Executable File
  File menu, Make EXE File
- Printing Code
  File menu, Print, select Code option
- Printing Forms
  File menu, Print, select Form option
- Saving a File and Naming It
  File menu, Save File As
- Saving a Form
  File menu, Save File

Note Saving a file does not mean that the project is also saved.

- Saving a Project
  File menu, Save Project

- Saving a Project and Naming It
  File menu, Save Project As

Note A Walk Through showing you how to save a text version of the code is located at the end of this portion of the module.

Editing Visual Basic Code: The Edit Menu

- Searching Code for a Text String
  Edit menu, Find
- Searching and Replacing a Text String
  Edit menu, Replace
- Cutting and Pasting Text
  Edit menu, Cut, Copy, and Paste
- Undoing Changes
  Edit menu, Undo

Testing Applications During Development: The Run Menu

- Starting the Application
  Run menu, Start

  Notice the change in the title bar when you select this option to indicate that you are now in Run mode.
- Stopping the Application
  Run menu, End

Visual Basic Window Management: The Window Menu

- Displaying the Properties Window
  Window menu, Properties
- Displaying the Toolbox
  Window menu, Toolbox
- Displaying the Project Window
  Window menu, Project Window
- Displaying the Color Palette
  Window menu, Color Palette

Getting More Information: The Help Menu

- Using the Help Table of Contents
  Help menu, Contents
- Searching for a Specific Topic
  Help menu, Search
- Locating Product Support Information
  Help menu, Product Support

Walk Through — Saving a Text Version of Code

Procedural programmers are used to seeing all of their code in one place. Remember, however, that the code is not executed in the order it appears in the text file you create here. It is still event-driven code, and the flow of execution still depends on user and system events.

∑ To save a text version of code

1. Restore Visual Basic.
2. From the File menu, choose Open Project.
3. Open WORLD.MAK, located in \WALKTHRU\SAMPLES.
4. Choose OK.

   Load the sample project that you've just completed, but do not run it.
5. From the Project window, select WORLD.FRM.

   Make sure that you select the correct file. When you load a project, the first file in the list is selected by default.
6. From the File menu, choose Save Text.

   Visual Basic will give your text file the same name as the form, in this case WORLD.TXT. That will probably do in most cases. Make sure that you save the file to the \WALKTHRU\SAMPLES subdirectory.
7. Choose OK.
8. Minimize Visual Basic.
9. Return to Program Manager.
10. In the Accessories group window, open Notepad.
11. From the File menu, Choose Open.

    Use the Open dialog box to locate WORLD.TXT.
12. Choose OK.

    This loads WORLD.TXT into Notepad so that you can review the code.

    If you want to print code directly to a printer, choose the Print command from the Visual Basic File menu.

    You can print just the form or code or both for the current file, or you can print all forms or code or both for your entire application. Remember, however, that .BAS files do not have a form associated with them, so in this case you will only be able to print code.

∑ To save a text version of properties and code

You can also save the form with the properties and code in a text file.

1. From Visual Basic, open the project WORLD.MAK in \WALKTHROUGH\SAMPLES.
2. From the Project window, select WORLD.FRM
3. From the File menu select Save File As.
4. Select the check box Save As Text and choose OK.
5. The form is saved in text format. The extension .FRM is used.
6. Use Notepad to compare the two text files.

Walk Through — Using Visual Basic Help

∑ To use Help

1. From the Visual Basic Help menu, choose Contents.

   The Visual Basic Help window appears. This window provides a brief topical tour of the major components of Visual Basic.

   A topical list that might be of interest to you is Programming Language.

2. Choose the Programming Language topic.

   Information on that topic appears in a window.

3. Choose the Beep Statement.

   Language reference material on the purpose, use, and syntax of this statement appear on screen.

∑ To use the Search command

Like standard Windows Help, the Visual Basic Help system has a Search command on the Help screen. To use Search, simply type in the term you are looking for.

1. From the Visual Basic Help window, choose Search.
2. Type **Toolbox** in the text box.

   The text box is located at the top of the screen.

3. In the Search dialog box, select Show Topics.
4. Select the topic Toolbox.
5. Choose Go To.

   Visual Basic opens a window with information on the term "Toolbox."

6. From the Help screen, choose Back.

   Visual Basic returns you to earlier topics.

∑ To use the History command

Use the History command to return to topics that you have covered earlier in the session. Topical lists in this option are arranged in reverse chronological order.

1. From the Visual Basic Help window, choose History.

   The Windows Help History dialog box appears.

2. Double-click the topic of your choice.

   The Help screen for that topic appears.

Σ To cut and paste sample code into your code

The most powerful part of the Visual Basic Help system is the large number of code samples that you can paste into a project and run.

The following procedures allow you to copy and paste sample code for a Click event.

1. Visual Basic should already be running.

   Make sure there is a new form on the screen.
2. From the Help menu, choose Search.
3. Type **Click** in the text box.
4. Click the Show Topics button.

   Click Event should be highlighted.
5. Choose Go To.
6. At the top of the Help form, click the word "Example."

   This brings up sample code in a window.
7. From the Event_Click example window, choose Copy.

   This brings up another window with all the sample code in it.
8. Highlight the code you want.

   In this case, take only the lines starting `Picture1.Move...`
9. Click the Copy button.

   This copies the selection to the Clipboard and hides the copy form.
10. Exit Visual Basic Help.
11. Double-click the picture box tool at the top of the Toolbox.
12. Drag the picture box to the lower-left corner of the screen.
13. Double-click the picture box on the form.

    The Code window will open.
14. From the Edit menu, choose Paste.

    Paste the sample code from the Clipboard to your form.
15. From the Run menu, choose Start.
16. Click the picture box once.

    It will move toward the upper-right corner of the screen.
17. Continue clicking the picture.

    It will disappear into the upper-right corner.
18. From the Run menu, choose End.
19. Minimize Visual Basic.

Summary

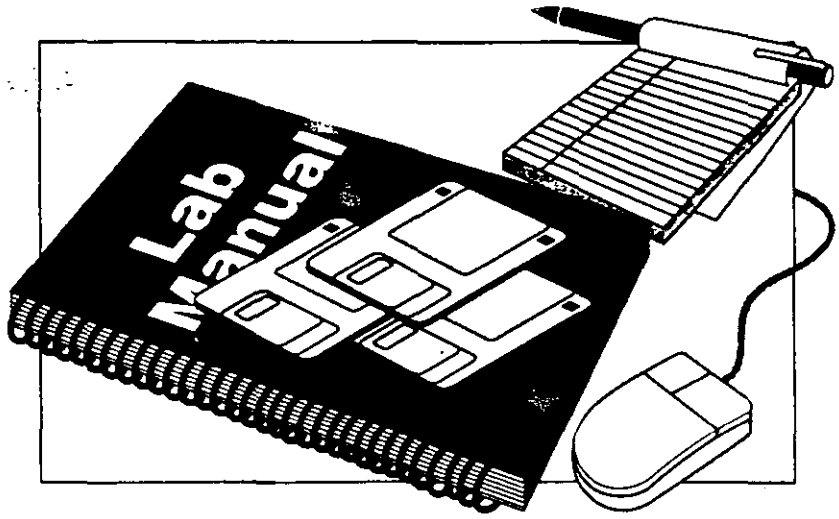
- **Visual Basic Tools**
- **Building a Simple Visual Basic Application**
- **Visual Basic Menu Commands**

Objectives

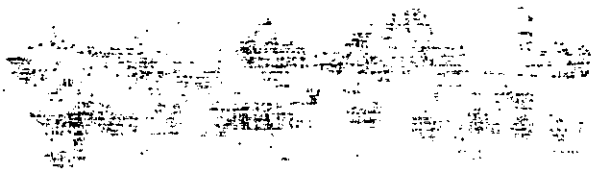
In this module, you learned to:

- Design an application user interface using the Form window, Toolbox, toolbar, and Property window.
- Name and save the application's forms and project files.
- Start and stop an application from the Visual Basic menu and/or the toolbar.
- Create an executable file and add it to a Windows group.

Lab Time



Go to the Creating An About Box portion of your lab manual.



Module 2: Designing and Building Visual Basic Applications

Overview

- Event-Driven vs. Procedural Programming
- Microsoft Visual Basic Terminology
- Application Development Process
- User Interface Design Guidelines
- Configuring Your Environment

Overview

The purpose of this module is to introduce you to several related concepts that help you make the transition from the procedural world to the event-driven world. This module creates the logical framework for much of the rest of the course. It contrasts programming for Microsoft Windows with MS-DOS and other character-based applications.

It also provides a high-level overview of and establishes relationships between objects and events.

This module also outlines the general process that is used to develop Visual Basic applications and some general suggestions for overall application design.

Prerequisites

None.

Overall Objective

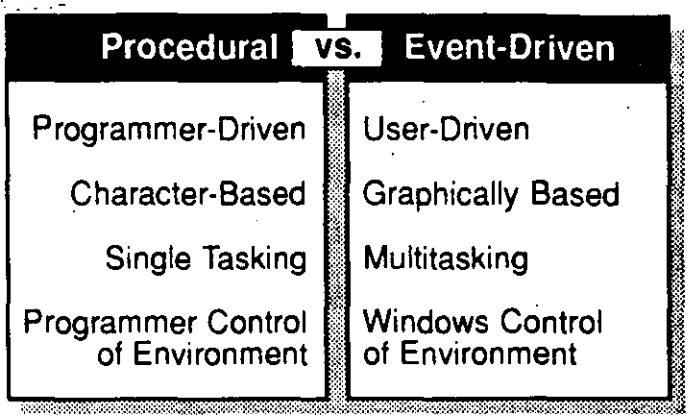
At the end of this module, you will understand the paradigm shift from procedural to event-driven programming.

Learning Objectives

At the end of the module, you will be able to:

- Explain the key differences between graphical and character-based applications.
- Provide high-level definitions for some key Visual Basic terms.
- Outline a basic application design and development procedure.
- Explain several fundamental principles of user interface design.

Event-Driven vs. Procedural Applications



Traditional programming is linear. It has a top-down sequencing that is controlled by the programmer.

Windows-based programming is event-driven. Windows-based events can be triggered in one of two ways. They can be either user-triggered or system-triggered.

Below, in pseudocode, is a very general summary of the structure of an event-driven program for the Windows operating system:

Example

```
Begin MAIN PROGRAM
    Begin Loop
        ' Ask Windows to pass messages to your program
        ' about what events have occurred
        Case Statement GETEVENT
            CASE Click
                ' You can choose to insert code to respond to
                ' click events for the appropriate objects.
            CASE Change
                ' You can choose to insert code to respond to
                ' change events for the appropriate objects.
            CASE ...
              .
              .
              .
            CASE Default:
                ' VB will handle other events internally
                ' without giving your program access.
                ' One of the default cases is the "end
                ' application" message that causes an exit
                ' from the loop and causes your application
                ' to terminate.
        End Case Statement
    End Loop
End MAIN PROGRAM
```

References

For a complete list of the events that Visual Basic recognizes, see the Introduction in the *Microsoft Visual Basic Language Reference*.

Notice that there is not a linear flow to the program because the flow depends on what events are generated and the order in which they are reported to your program by the Windows operating system.

Applications that run in the MS-DOS environment are programmed to be the only application running. MS-DOS–based applications do not handle multitasking very well.

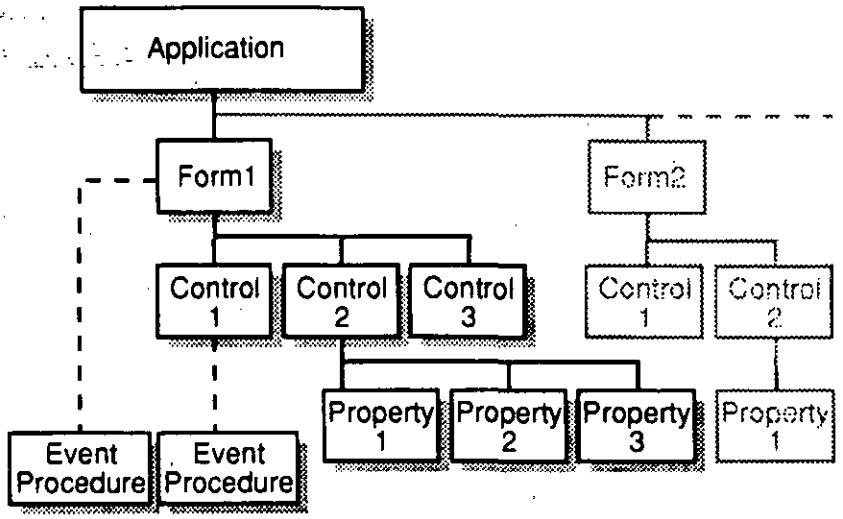
Window applications are multitasking. They can share screen space and computing time.

MS-DOS–based applications are character-based. Windows-based applications are graphically based and typically use proportionally spaced fonts.

Applications programmed to run in the MS-DOS environment are able to control the environment the user operates within. The program can have control over the sequencing and appearance of where the user will go next in the application.

Applications programmed to run under Windows give control of the environment to Windows. How the user moves between events is controlled through the Windows operating system itself rather than through the Windows-based application.

Visual Basic Terminology



The terms on this foil depict two major components of the Visual Basic development environment: the graphical side of it and the code side. Put in tabular form, they look like this:

Application
- Forms
  - Properties
  - Events
- Controls
  - Properties
  - Events

What Is an Object?

Consider this example: The dashboard of your car offers users a variety of gauges, dials, and accessories that take input and give output. The thermostat tells you how hot the engine is. The steering wheel lets you change direction. Each one of these "objects" performs a specific purpose, and you use it to attain a given goal or objective. Users learn how to use these objects.

For the most part, users do not learn how to install, maintain, or troubleshoot these things. They do not learn how these things function. The user only needs to know that these objects work and what to expect from them. In a like manner, Visual Basic offers programmers objects—forms and controls—that they can use to build applications. For the most part, the programmer need only use the object without necessarily having a detailed understanding of how the object does what it does.

The Graphical Side

Objects

Components of an application, usually forms and controls.

Forms

Forms are the building blocks of applications. They are the windows that users see when they run your application; they are the major structural units that make up your application. Visual Basic is made up of several forms. The Toolbox, the Properties window, the Project window, and the Save Project As dialog box are all examples of forms.

Walk Through — Forms and What You Get Free

To open a blank form

1. Start Visual Basic.

   A blank form should be on the screen.

2. From the Run menu, choose Start.

   Notice all of the things that Visual Basic gives your form, such as:

   - Sizing border
   - Control menu
   - Minimize button
   - Maximize button

   You don't need to write any of the code for painting any of these features of your application. You also don't need to write any of the code for managing these features.

3. From the Run menu, choose End.

Controls

Controls are the tools you place on forms that provide users with application functionality. Examples of controls are the command buttons and labels.

Properties

Forms and controls have properties (attributes) that you can change during design and run time. An example of a property is the caption that appears on a command button. Generally speaking, you set the values for control and form properties when you are designing your application.

Event Procedures

Event procedures are code internal to Visual Basic and Windows that is written for you and provides your application with some of its basic functionality. For example, when a user clicks a check box, the Click event knows how to paint an X in the square. However, you must place code inside the Click event for that check box to cause your application to react appropriately when a user places an X in or removes an X from the check box.

The Code Side

Project

Your development project is made up of more than graphical forms and controls; it also has Basic code in it. This is managed and accessed through the Project window.

Walk Through — Visual Basic View of Your Project

To view your project

1. If it isn't already started, Start Visual Basic.
2. From the File menu, choose Open Project.
3. Load the ICONWRKS.MAK file.

   This file is located in \VB\SAMPLES\ICONWRKS.
4. From the Run menu, choose start to run the application.

   Work with the application for a minute, but the point here is that this application is made up of a number of different forms.
5. From the Run menu, choose End to stop the application.
6. Access the Project window.

   The Project window keeps track of the forms and modules that make up your application.

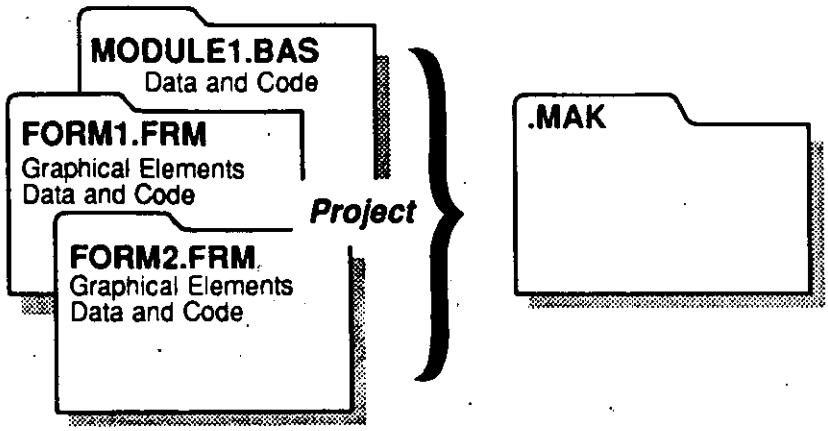
   For our purposes, Visual Basic treats each of your forms as a separate component.

   Visual Basic keeps track of the number and kinds of files that you are using in your application in the .MAK file and loads them when you want to start working on your project.
7. Quit Visual Basic.

Module

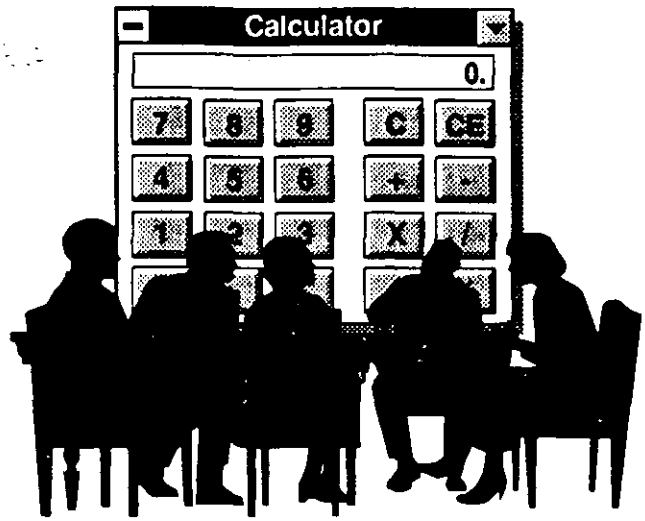
In Visual Basic a module is a file that contains only code. One example of a .BAS file that you could see at the top of a Project window is MODULE1.BAS.

Another way of showing the relationship among forms, controls, properties, and event procedures is this:



The .MAK file contains names of both the .FRM and the .BAS files in the project.

Application Development



Creating an Application

To create an application in Visual Basic, follow this suggested sequence

1. Open a new project (or use the new project created when you start Visual Basic) to organize the parts of your application.
2. Create a form for each window in your application.
3. Draw the controls for each form.
4. Create the menu bar for the main form.
5. Set the form and control properties.
6. Write event procedures and general procedures.
7. Save your work.
8. Debug your code.
9. Create an executable file to turn the project into an application.

Distributing the Application

When you distribute the executable of your application to users, you need to distribute a copy of the Visual Basic dynamic-link library VBRUN300.DLL with it. This dynamic-link library (DLL) is a part of the Visual Basic installation files, and you can distribute it royalty free. If you build your application using any of the available custom controls, you will also need to ship all appropriate .VBX files, and in some cases .DLL files. The product documentation for the custom controls will tell you what these files are.

Finally, during design time, you may decide to move the forms for the application to another machine. In this case, you will probably need to rebuild the .MAK file, because it keeps track of all the files that make up your application as well as the fully qualified path to them. To do this, copy all the files to the new machine, then update the paths in the .MAK file by using the Remove File command from the File menu, and then the Add File command.

User Interface Design

General Guidelines

- Design Basics
- Color
- Fonts

User interface design guidelines are an agreement to create consistent user interfaces.

User interface guidelines are important for ease of learning by the user. They also prevent programmers and developers from "reinventing the wheel."

They are guidelines—suggestions on the way you might want to design the user interface. There are no hard and fast rules.

General Guidelines

- Design basics
  - Design for the user, not the system

    Composition and functionality
  - User control
  - Directness
  - Consistency
  - Clarity
  - Aesthetics
  - Feedback
  - Forgiveness
- Color
  - Color as an attention-getter
  - Complementary versus circus colors
- Fonts
  - Serif versus sans-serif
  - Size
  - Number (variety)

For More Information

For more details on user interface design guidelines, see:

- *The Windows Interface: An Application Design Guide*
- *Visual Design Guide* contained within Visual Basic

Walk Through — Designing the User Interface

As you walk through this application, look at the property listed and answer the question that follows.

Σ Print Utility #1

1. From the Walk Throughs program group, start Print Utility #1.

   First impressions are important. Are the function and purpose of the application apparent from a first look at the interface?

   _____

2. Title Bar

   The title bar should contain the name of the application. Does it?

   _____

3. Menus

   The menu structure should reveal something about the contents of the application. Does it?

   _____

   Does the first menu item follow user interface guidelines?

   _____

   Is the Help command in the standard place?

   _____

   Is the Quit command handled suitably?

   _____

   Where does the About command normally appear?

   _____

4. Scroll bar

   Is this the most effective way to ask users for the number of copies they will be wanting?

   Are the size and location of the scroll bar appropriate given the overall purpose of the application?

   ---

5. Options

   How many sets of options are there really on this form?

   Are the check boxes presented in red, the best way to get information from the user?

   How should the header, footer, and page numbering questions be implemented?

   ---

6. FileName text box

   If the user wants to find a file and print it, what other tools does the user need besides this simple text box?

   ---

7. Stop and Print buttons

   Are these buttons appropriately sized for this form and its function?

   ---

8. Exit

   Close the application.

Σ Print Utility #2

1. From the Walk Throughs program group, start Print Utility #2.
2. Open the second version of this application.

   Is the overall layout of the form effective? How does the layout suggest how the user might work with the form?

   _____

   How do the frames help structure the user's decisions?

   _____

   How is the menu structure consistent with user interface guidelines?

   _____

   Why were the header and footer options dropped from the form?

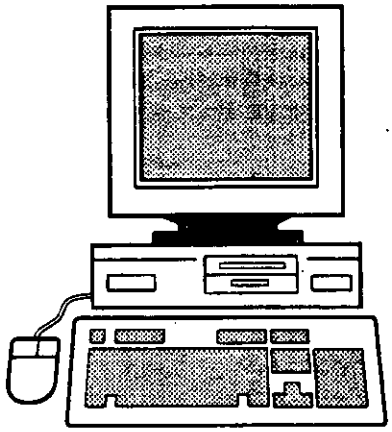
   _____

   Why was the Percent Done label added?

   _____

3. Quit the application.

User Interface Design Guidelines



Device Input

Getting Input from a Mouse

There are a limited number of things that users can do with a mouse:

1. Single-click with either the primary or secondary (left or right, top or bottom) button.
2. Double-click with either the primary or secondary button.
3. Drag normally with the primary button.
4. Drag and drop.

For example, a single click with the primary mouse button normally indicates that a choice such as a bold, italic, or underline text format option has been made on the ribbon of a text editor. Selection of a file to be opened is normally indicated by a single click.

Double-clicking selects the object to be acted upon and initiates the action. That is, if you want to select and open a file, you simply double-click a filename. The same thing happens when you double-click an icon.

Dragging is used to resize a window, move the window to a new location, or reposition an icon. Drag and drop is normally used to select an object, for example, the name of a file, and then place it over another object, for example, a printer, so that the file can be printed.

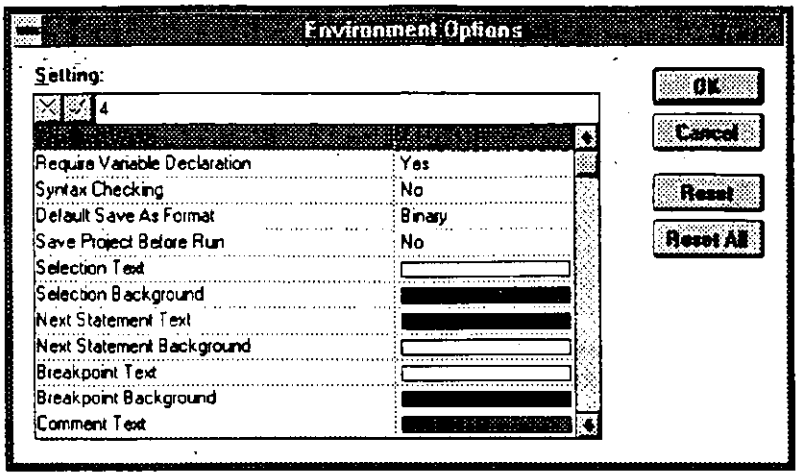
Keyboard and Mouse Functions

Whenever possible, all mouse actions should be duplicated with an equivalent keyboard action. Keyboard shortcuts and access characters should also be used to reduce the number of keystrokes and potential for errors by the user.

For example, in most cases you will want to give a keyboard shortcut for the most commonly used commands. If possible, use the first letter of a menu or control as the access character. If there is a conflict, use another letter in the menu or control name.

When you design the application and its forms, specify the shortcut keys.

Configuring Your Environment



You can configure much of the look and feel of the development environment. From the Options menu, choose Environment Options. From this window you can set everything from the default tab stop in code to the foreground and background colors of comments in code.

Setting	Default
Tab Stop Width	4
Require Variable Declaration	No
Syntax Checking	No
Default Save As Format	Binary
Save Project Before Run	No
Selection Text	
Selection Background	
Next Statement Text	
Next Statement Background	
Breakpoint Text	
Breakpoint Background	
Comment Text	
Comment Background	
Keyword Text	
Keyword Background	
Identifier Text	

Setting	Default
Identifier Background	
Code Window Text	
Code Window Background	
Debug Window Text	
Debug Window Background	
Grid Width	120 Twips
Grid Height	120 Height
Show Grid	Yes
Align to Grid	Yes

For a list of the colors available, see the Colors section of CONSTANT.TXT located in the \VB subdirectory.

Note Changing to a larger grid size and then aligning the controls to the grid may change the size of your controls.

Automatically Loading Visual Basic Extensions at Startup

You can control which of the .VBX or Visual Basic Extensions are loaded when starting up a new project by editing AUTOLOAD.MAK with a text editor. This file is located in the \VB subdirectory. The default settings are:

```
GRID.VBX
MSOLE2.VBX
ANIBUTON.VBX
CMDIALOG.VBX
CRYSTAL.VBX
GAUGE.VBX
GRAPH.VBX
KEYSTAT.VBX
MSCOMM.VBX
MSMASKED.VBX
MSOUTLIN.VBX
PICCLIP.VBX
SPIN.VBX
THREED.VBX
ProjWinSize=152,402,248,215
ProjWinShow=9
```

Summary

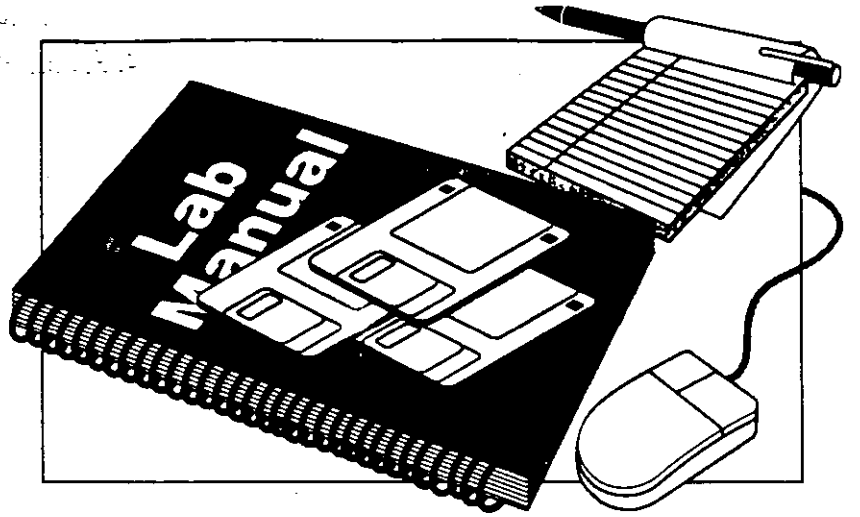
- Event-Driven vs. Procedural Programming
- Microsoft Visual Basic Terminology
- Application Development Process
- User Interface Design Guidelines
- Configuring Your Environment

Objectives

In this module you learned to:

- Explain the key differences between graphical and character-based applications.
- Provide high-level definitions for some key Visual Basic terms.
- Outline a basic applications design and development procedure.
- Explain several fundamental principles of user interface design.

Lab Time



Go to the Employee Database Application Specification portion of your lab manual.

Module 3: Working with Forms

Σ Overview

- Forms and Their Properties
- Message Boxes
- Starting the Forms of the Employee Database
- Multiple Document Interface Applications

Overview

The purpose of this module is to introduce you to the Visual Basic programmer's fundamental tool—the form.

Even though this is a relatively brief module, it introduces some key terms that are required for a full understanding of the programming environment. This module primarily focuses on the various properties associated with forms, separate from the discussion of controls and properties. This module also touches on event procedures and methods associated with forms. A more detailed discussion of these issues will be found in a separate module.

This module also serves as the introduction to the main elements of the class application and walks you through creating, naming, and saving these forms.

Prerequisites

Prior to starting this module, you should have a fundamental awareness of:

- Windowing technology from the user perspective
- The Visual Basic programming environment

Overall Objective

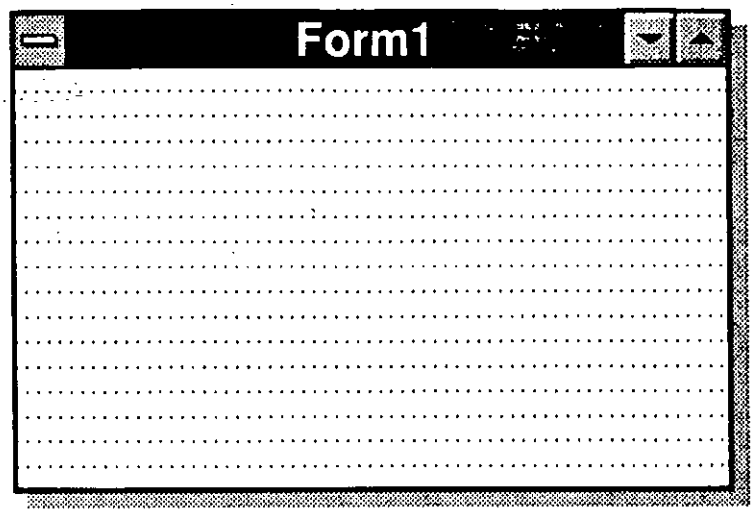
At the end of this module, you will understand the fundamentals of working with forms in Visual Basic.

Learning Objectives

At the end of this module, you will have:

- Set captions on forms.
- Set the Name property for a form.
- Added a form to a project.
- Saved all of the forms in a project.
- Saved the project itself.

Properties for Forms



Forms and Their Common Properties

Forms are the central element of Visual Basic. They are the design of your application; they are the thing that your user interacts with. They are where you place the controls for your application.

You will notice that many of the properties for forms have the same names as those found for controls, but on forms they have a different use.

Property	Default	Comments
BorderStyle	2 - Sizable	Set BorderStyle to "3 - Fixed Double" to create a modal form (used for dialog boxes).
Caption	Form1	Appears at the top of the form.
ControlBox	**True**	Enables the Control menu.
FontSize	8.25	Used in printing to a form.
FontName	Helv	Used in printing to a form.
Name	Form1	The name in code. Also appears on Project list.
Height	4425 twips	
Icon		Default icon for your executable.
KeyPreview	**False**	Determines whether the form keyboard events occur before control keyboard events.
Left	–	From left edge of screen.
MaxButton	**True**	Disable for a modal dialog box.

Property	Default	Comments
MinButton	**True**	Disable for a modal dialog box. Also see BorderStyle.
MousePointer	0 Default	Sets cursor shape on form—13 possible choices. See the *Microsoft Visual Basic Language Reference*.
Top	–	From top edge of screen.
Visible	**True**	False is equivalent to calling the **Hide** method.
Width	7485 twips	

Events

Load A Load event occurs when a form is loaded. Normally, you will use a Form Load event to set initial values for the controls on a form.

Example

```
Sub Form_Load ()
   Top = 1500
   Left = 1000
End Sub
```

Unload An Unload event occurs when a form is about to be removed from memory. This event is normally triggered by the user closing the form using the Control menu.

Methods

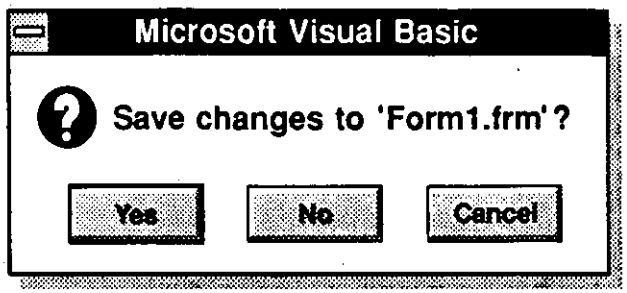
Hide This method is used to remove a form from the screen without unloading it. When you use the **Hide** method, the Visible property of the form is set to **False**.

Show The **Show** method is used to display a hidden form.

Example

```
Sub mnuPrintOptions_Click ()
   frmPrintOptions.Show
End Sub
```

Message Boxes



What Are Message Boxes?

For the most part, Visual Basic applications are made up of forms. You will find several places in your applications where you want to convey fairly routine kinds of information to users and you really don't want to design and build a form to deliver it.

Visual Basic has a convenient tool that displays a kind of form—called a message box—where all you need to do is to provide the message string and, optionally, a number that indicates the number and types of buttons and icons to be displayed on the form, and a title for the dialog box.

If you wanted to make sure that users had saved all of their new data prior to closing an application that you are writing, you could create a whole new form, or you could use the **MsgBox** function.

Walk Through — Coding Message Boxes

To inspect the message box sample application

1. From the Walk Throughs program group, start MsgBox.
2. Click the Exit button on Form1.

   This displays a dialog box that queries users about the status of their data.
3. Choose Yes, No, or Cancel.

   Any one of these choices closes the dialog box. Implementing code that differentiates them will be added in another module.
4. Open the Control menu on Form1, and choose Close.

   This closes Form1.

Σ To code the message box sample application

1. If Visual Basic is not running, start it.
2. Double-click the command button tool in the Toolbox.
3. Place it in the lower-right corner of the form, or any other out-of-the-way place.
4. In the Properties window Properties list for the command button, change the following property:

   Caption: Exit
5. Click anywhere on the form.

   This puts the focus back on the form.
6. Double-click the command button.

   This will automatically open the Code window for the form and take you to the Command1_Click event template.
7. Add the following code:

```
Msg$ = "Have you saved all your work?"
MsgBox Msg$, 3 + 32, "MsgBox Walk Thru"
```

   What does the code do? First you create a string variable with the message you want to display, and then you call the **MsgBox** statement. You can pass the **MsgBox** statement three arguments. Although the second and third arguments are optional, in this case we have you include them.

   The first argument is the message string that is displayed inside the message box.

   The second is a sum of values that indicates to Visual Basic the type of message box you want. More about these numbers in a minute.

   The third argument is the title of the message box.
8. From the Run menu, choose Start.
9. Test the dialog box.

   Use the mouse pointer to choose the Exit command button. The dialog box will appear in the middle of your display.
10. To close the Message WalkThru dialog box, choose the Cancel button.
11. From the Run menu, choose End.
12. In the Code window, highlight MsgBox.

    Use the mouse pointer to highlight this word in the code.

13. Press F1.

    Make sure you are in the design mode, not the run mode.

    By adding the appropriate key values listed in the second table in the Visual Help topic for the **MsgBox** statement, you can control the number and type of buttons, control the icon displayed in the message box, and even set a default button.

    In this walk through, you add the value 3 (to request a message box with Yes, No, and Cancel buttons) and the number 32 (to request a message box with a Warning Query or question mark icon).

    There are more sophisticated ways of implementing and using message boxes, but this gives you a start.

    Did you notice that the Yes, No, and Cancel command buttons do much, yet?

    If you read further down in the Help window, you see that each of the different buttons returns a different value. You will use those values a little later in the course, along with some conditional logic to code the three buttons on the dialog box. For the time being...

14. Minimize Help.
15. From the File menu, choose Save File As.

    Save this form in \WALKTHRU\STUDENT1 as MSGBOX.FRM.

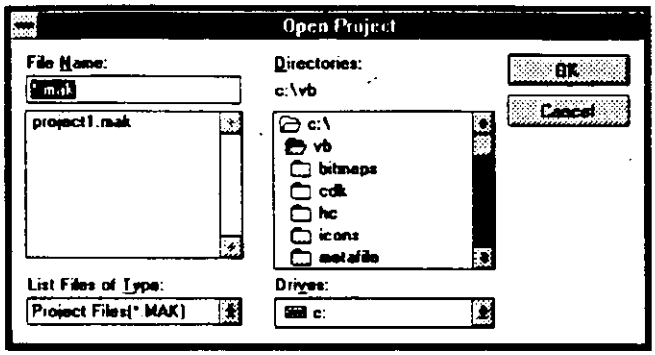
16. From the File menu, choose Save Project As.

    Save the project in the same subdirectory as MSGBOX.MAK.

17. Select the Visual Basic Control menu, and choose Close.

    Quit Visual Basic and Help.

Modal and Modeless Dialog Boxes



Modal, Modeless, and System Modal

You can designate a form as being either modeless or modal. The default is modeless, which means that the user can open the form and still get to other forms within the application to do work. Modal forms, in contrast, require that the user do something — click a button, check to make sure they really do want to delete all those files — before they can do any work.

Walk Through — Modal Dialog Boxes

To use a modal dialog box

1. From the File menu, choose Open Project.

   Bring up the Open Project dialog box.

2. Access the Project window.

   Place the cursor on any other portion of the Visual Basic interface. Press either one of the mouse butt

   A beep sounds and the cursor continues to flash in the text box. In this case the user must choose one of the three choices presented: Locate the name of a project using the list boxes, input a valid project name, and choose OK or Cancel.

3. Click the Cancel button.

   Close the dialog box.

Application Modals and System Modals

Modal dialog boxes do have a limit; they only guide the user within their own application. They have no effect if the user switches to Windows or to another application. In order to make a form system modal, all you need to do is to use the *style%* parameter for both **MsgBox** statements and functions.

The syntax looks like this:

[form.] **Show** [*style%*]

In addition, you would need to declare three global constants:

```
Global Const MODAL = 1
Global Const MODELESS = 2
Global Const SYSTEMMODAL = 4096
```

Walk Through — Making the Forms of the Employee Database Application

To make the forms of the Employee Database application

1. If Visual Basic isn't already running, start it.
2. Set the following properties for the form:

Caption	Employee Database
Name	frmEmpDB
Height	5760 (approximately)
Icon	\VB\ICONS\MISC\MISC28.ICO
Width	7600 (approximately)

Note When you set the Icon property for a form, you will need to know the full path to the source file, but the Visual Basic interface will only tell you that you have an icon by displaying the property as (Icon).

3. From the File menu, choose Save File As.

   Make sure that you save the file in the appropriate subdirectory. If you don't, the .MAK file will not be able to find this part of your project. This first pass at the application should be saved in \STUDENT1\FORMS.
4. Name the file EMPDB.FRM.
5. Choose OK.

   This saves the form.
6. Click the Project window.

   Notice that the filename is in the left column on this list, and the name of the form is in the right column.

   So far, you have given the form the names that are known to the file system and to Visual Basic.
7. From the Control menu for the Employee Database form, choose Close.

   Close EMPDB.FRM.
8. From the File menu, choose New Form.

   Add a second form to the application.

9. Set the following properties for the form:

Caption	Employee Record
Name	frmEmpRec
Height	4545 (approximately)
MaxButton	False
Width	5485 (approximately)

10. From the File menu, choose Save File As.

    Now, rename and save the file, making sure that you save it in \STUDENT1\FORMS.

11. Name the file EMPREC.FRM.
12. Choose OK.
13. From the Control menu on your newly created form, choose Close.

    Close the form, and notice that the name of this form has been added to the list in the Project window.

    User interface guidelines suggest that applications have an About box—a form that indicates that the application is copyrighted. You have already created an About box; now you need to add it to this application.

Note You will also want to make some changes to the About form, but the directions for doing this are not included here. You would, for example, change the caption for the About box so that it contained the name of the application, but doing that here distracts from the point of this walk through.

---

14. Copy ABOUT.FRM from \STUDENT1\ABOUT to \STUDENT1\FORMS.

    Before you add the About box form to the project list, you should make a copy of the About box form that you have already completed and place that in \STUDENT1\FORMS.

15. From the File menu, choose Add File.

    ABOUT.FRM should appear in the \FORMS subdirectory, so all you need to do is select the name so that it appears in the text box.

16. Choose OK.

    The name of the added form should appear in the Project list.

17. From the File menu, choose Save Project As.

    Now that you have made most of the basic forms for the Employee Database application, you need to rename and save the file that keeps track of the files in the project.

    The Save Project As dialog box should appear center screen, with the correct subdirectory already listed in the current working directory and a suggested filename with the appropriate file extension.

18. Name the project as EMPLOYEE.MAK.
19. Choose OK.

    This saves all of your current build information.

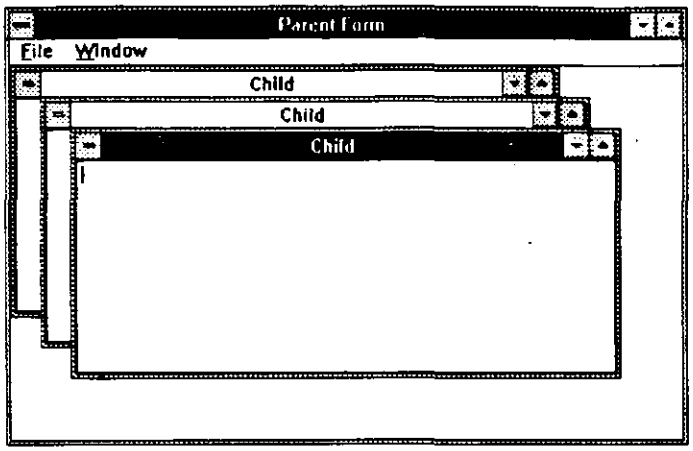
20. From the Run menu, choose Start.

    If you want to test the application, you should see the Employee Database form appear on screen. This form appears only because that was the first form you created. If you try to open the Employee Record form, you will see that you have not-implemented code for that task yet.

21. From the Run menu, choose End.

22. Quit Visual Basic.

Multiple Document Interface Applications



Placing Forms Within Forms

Visual Basic allows you to write applications that can create multiple copies of a form and display all the forms within a single container form. Microsoft Word for Windows and Microsoft Excel are both applications that allow users to do this.

Walk Through — Creating MDI Applications

To see the final version of your MDI application

1. From the Walk Throughs program group, start MDI.

   For the most part, in this course, we will focus attention on placing controls on forms. However, Visual Basic has the capacity to place forms on forms and multiple instances of forms on forms.

   The purpose of this walk through is to give you the fundamentals of implementing an MDI application. We don't include all the enhancements that you might want to implement. Detailed coverage can be found in the follow-on *Programming in Microsoft Visual Basic 3.0* course.

2. From the File menu, select New.

   Do this three times. You may not see much happening at first, but each time you select New, another instance of the child window is being drawn on screen, one atop the other.

3. From the Window menu, choose Tile.

   This organizes and displays all of the child windows that you have created.

4. From the Control menu on the Parent form, choose Close.

   In order to create this application framework, follow the steps below; but there is a warning required here. You will be typing in a number of things that haven't been explained yet. Don't worry; they will be. For the time being, accept that things work and that the details will follow.

Σ To create a simple MDI application

1. Start Visual Basic.

   A new, blank form will appear on screen.

2. From the File menu, choose Add File.

3. Use the browser to add PARENT1.FRM located in \WALKTHRU\FORMS.

Note Under normal circumstances, you wouldn't go this route. You'd select the New MDI Form command. If you did that, a second form would appear on screen, the entry named MDIFORM1.FRM would be added to your project list, and the New MDI Form command on your File menu would be disabled. From there, you would need to add all the menu items; but since you haven't done that yet, we will give you the completed form, so that you can concentrate on the MDI capabilities.

   Now you should have two forms in your project.

4. In the project list, put the focus on FORM1.FRM.

   That is, make sure that you see FORM1.FRM on top of all the other forms within Visual Basic.

5. From the File menu, choose Save File.

   Save the file as FORM1.FRM.

6. From the Window menu, select Properties.

   This displays the Properties window for FORM1.FRM.

7. In the Properties window Properties list, locate the MDIChild property.

   The default value for this property is **False**.

8. Double-click the value in the table.

   This toggles the value to **True**.

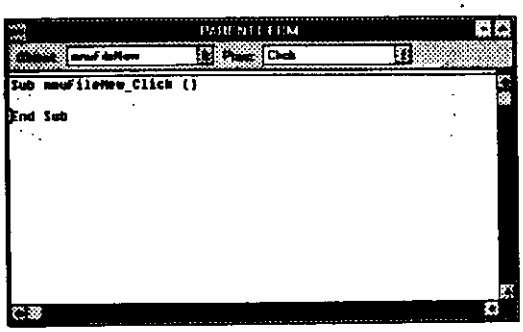
9. Select the Control menu for the Properties window and choose Close.

10. In the Project window, highlight PARENT1.FRM and choose View Form.

    This places the focus on this form.

11. Open the File menu on the Parent Form and choose the New command.

    This is the shortcut for getting from the form to the code that supports it. You should now be in a Code window that looks like this.



12. Place your cursor on the line between the **Sub** and **End Sub** lines and add this code:

```
Dim NewDoc As New Form1
NewDoc.Show
```

That's all there is to it. Your questions about **Dim** and **New** and **Show** will all be answered in a little while.

13. From the Object drop-down combo box, choose General.

This moves you to the General Declarations section of the code, where you can declare a couple of constants that Visual Basic needs for arranging the windows that your application creates.

14. Add the following code:

```
Const CASCADE = 0
Const TILE_HORIZONTAL = 1
```

15. From the Object drop-down combo, select mnuWindowCascade.
16. Add the following code on the blank line between the **Sub** and **End Sub** lines:

```
MDIForm1.Arrange CASCADE
```

17. From the Object drop-down combo, select the mnuWindowTile.
18. Add the following code on the blank line:

```
MDIForm1.Arrange TILE_HORIZONTAL
```

That should do it. Now all you need to do is to run your application.

Σ To run your new application

1. Choose the Run icon on the toolbar.
2. From the File menu on the Parent form, choose New.
3. Do this a couple of times so that you have several windows to work with.
4. From the Window menu on your application, choose Tile.

   This tiles all of your windows.
5. From the Window menu on your application, choose Cascade.

   This re-arranges all of the windows.
6. From the Run menu, choose End.
7. From the Visual Basic File menu, select Save Project As.

   Now that you have the framework for an MDI application finished, save your work.
8. Save the project as MDI.MAK and make sure that it is in \WALKTHRU\SAMPLES.

Summary

- Forms and Their Properties
- Message Boxes
- Starting the Forms of the Employee Database
- Multiple Document Interface Applications

Objectives

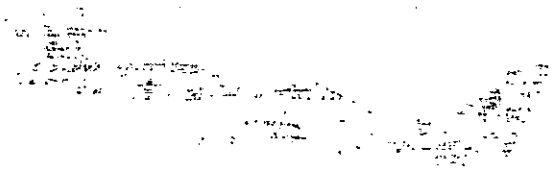
In this module you learned to:

- Set captions on forms.
- Set the Name property for a form.
- Add a form to a project.
- Save all of the forms in a project.
- Save the project itself.

Module 4: Laying Out Menus

Overview

- Menu Guidelines
- Microsoft Visual Basic Implementation



Overview

Previous modules gave you a chance to see and use the processes you should follow for developing Visual Basic applications. In those modules you developed a single-form application. In another module, you developed all the forms that are needed to implement the database front-end, but there was a part that was missing—the menus on some of the forms. This module begins with a discussion of the general layout of menus and the user interface specification. It ends with a demonstration/walk through of the Menu Design window.

Prerequisites

To successfully complete this module, you must be able to use a mouse. Experience with Windows-based applications is useful but not required. You should also be able to start Visual Basic and create a form.

Overall Objectives

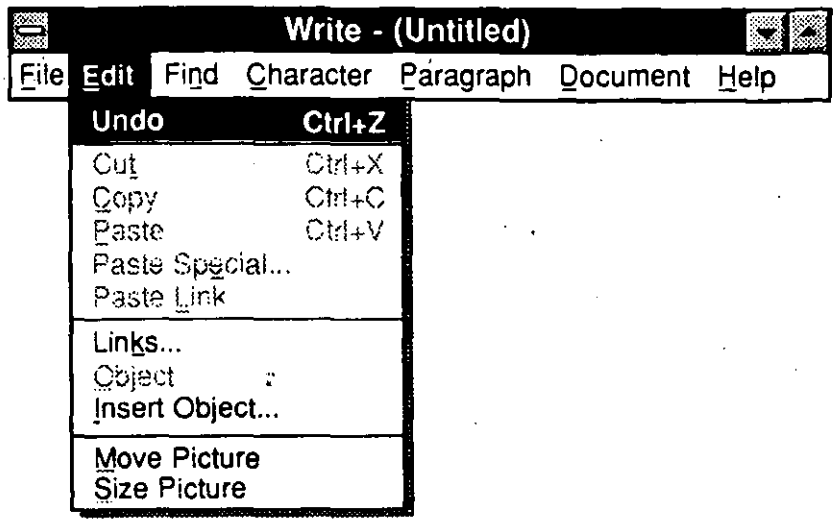
The overall objective of this module is to teach you how to design user-friendly menus and create them using Visual Basic.

Learning Objectives

At the end of this module's lab, you will be able to:

- List and explain the use and value of all of the key menu elements:
  - Access or hot keys
  - Shortcut keys
  - Menu bar
  - Separator bar
  - Ellipsis
- Create an application that uses at least one menu on an application window and uses more than one form. It should include a least one functionality menu as well as a Help menu that includes choices to display a simple Help form and an About box.

Menu Guidelines



Menus and Their Standard Properties

Under user interface guidelines, menus are one of the primary ways for programmers to structure application functionalities for users. Normally, menus do one of two things: explicitly invoke a command, such as closing an application using Exit, or invoke a dialog box that offers users more options, such as the bold and italic text-formatting options.

Menu Structure

Under normal circumstances, most applications will start out life on the development side with three main menus—File, Edit, and Help. The File menu normally will contain commands that are related to file manipulation:

- New
- Open
- Save
- Save As
- Print
- Print Setup
- Repaginate
- Exit

Style Guidelines

Quitting the application, by convention, is done from the bottom of the first menu. Help is normally the last menu on the right.

The general Edit menu normally contains at least these four commands:

- Undo
- Cut
- Copy
- Paste

It may contain several other commands as they are needed for specific functions within the application.

Finally, Help is normally implemented with at least these four choices:

- Contents — Table of contents for Help
- Search for Help on... — Index to Help
- How to Use Help — Directions for using Help
- About *application name* — Copyright notice

Special Features of Menus

Period Ellipses

The three dots at the end of a command on a menu indicate that a dialog box will appear offering you more choices related to the command. A lack of three dots indicates that as a result of choosing the command, an action will be carried out. For example, if you choose Centered from the Paragraph menu, all highlighted paragraphs will be centered on the page.

Checking Options

A check mark next to a menu item indicates to the user that the option is currently invoked.

Separating Clusters of Related Commands

Separator bars are used to visually cluster related commands so that the user sees a short list of related items, rather than a long list of seemingly unrelated items.

Access or Hot Keys

Access keys are marked by an underscore beneath a single letter in the menu item or command. Access keys are the keyboard (as contrasted with the mouse) input device for the menu. Access keys are invoked by pressing the ALT key and then the underscored letter in the command. For example, pressing ALT+F opens the File menu. Pressing ALT+F+O opens the File menu and then chooses the Open command.

Style Guidelines

For access keys, you should try to select the letter that will be most memorable for most users. Normally this is the first letter in the word, but that might not always work. For example, Minimize and Maximize both begin with "M," so a better set of options might be to use the "n" from Minimize and the "x" from Maximize for the access keys.

Shortcut Keys

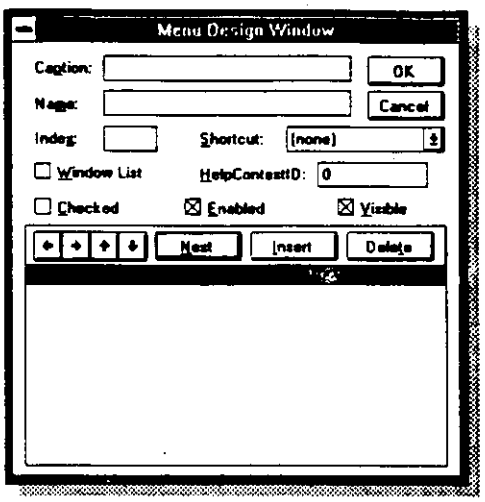
Shortcut keys are a second form of keyboard access to the menu. To use shortcut keys, the user need only press a function key or some other key combination (such as CTRL+A) in order to execute a command.

Style Guidelines

Some examples of standard shortcut keys are as follows:

F1	Help
CTRL+X	Cutting selected text
CTRL+C	Copying selected text
CTRL+V	Pasting selected text
CTRL+Z	Undo

Visual Basic Implementation



The Menu Design Window

From the Window menu, choose Menu Design window. If the Menu Design window command is unavailable (dimmed), click any of the forms in your Visual Basic application. You will then be able to access the Menu Design window. The Menu Design window dialog box appears.

The remainder of this topic divides the Menu Design window into two units—Menu Layout Items and Layout Manipulation Items.

Menu Layout Items

Item	Default	Description
Caption		Name that appears on the menu.
Name		Name used in code. Use **mnu** as a prefix.
Index		Used for adding menu items dynamically.
Shortcut		CTRL + shortcut keys.
Window List		Specifies whether a menu control will include a list of open MDI child forms.
HelpContextID		Specifies an identifier for the menu item in a Help system.
Checked	Not checked	Indicates whether a check mark will be displayed to the left of a menu choice.
Enabled	Checked	Indicates whether a menu choice is available to the user (disabled choices are dimmed).
Visible	Checked	Indicates whether a menu item is visible to the user.

Layout Manipulation Items

In order to use most of the Layout Manipulation Items on the Menu Design window, you must first select the control (menu) and then press the manipulation item to alter the control.

Item	Function
LEFT ARROW	Raises a menu item one level—for example, makes a submenu into a main menu.
RIGHT ARROW	Lowers a menu item one level—for example, makes a main menu into a submenu.
UP ARROW	Moves the menu item up one position in the menu list.
DOWN ARROW	Moves the menu item down one position in the menu list.
Next	Moves the cursor down one item in the menu list. Also, clears the Caption and Name so that you can add a new menu item.
Insert	Inserts a blank line in the list of menus so that you can add a new menu item.
Delete	Deletes the selected menu item from the list.

Style Guidelines

In order to implement access or hot keys, place an ampersand (&) in front of the letter to be used in the ALT+ combination. For example, to implement ALT+F as the access key combination for the File menu, you would type **&File** in the Caption control of the Menu Design window.

Separator bars are implemented by entering a single hyphen (-) as a Caption.

Note You must assign a menu name to the separator even though it is not really acting as a control.

Walk Throughs — Startup Application with Menus and Click Events

To see the final version of the menus Walk Through

1. From the Walk Throughs program group, start Startup.

   This sample application serves two purposes. It introduces you to the methods for implementing menus in Visual Basic applications, and it gives you a simple example of how Click events work in Visual Basic. Incidentally, this sample application also shows you how to use the **Shell** function.

2. Choose the Cardfile command button.

   This application starts up Cardfile or Windows Paintbrush™.

3. Close Cardfile.

4. From the Edit menu, choose Colors.

   The application shows the implementation of menus, cascading menus and checked menus. It also uses the ampersand for hot keys.

5. From the Colors cascading menu, choose Red.

   Finally it shows the use of a Click event to reset the background color of the application.

   Now, the question is: How did we do all of that?

6. Close the Startup application.

Σ To implement menus on the Startup application

1. If Visual Basic isn't running already, start it.

   A new blank form should be on the screen.

2. From the File menu, choose New Project.

   Start a new project.

3. Set the following properties for the form in the Properties window:

Name	frmStartup
Caption	Startup Applet
Height	2860 (approximately)
Width	2670 (approximately)

4. Open the Menu Design window.

5. Make the following changes.

Menu item	Caption	Name	Indentation
Edit	&Edit	mnuEdit	0
Colors	&Colors	mnuColors	1
Red	&Red	mnuRed	2
White	&White	mnuWhite	2
	Select the Checked option.		
Blue	&Blue	mnuBlue	2
-	–	mnuSep1	1
Exit	E&xit	mnuExit	1
Help	&Help	mnuHelp	0
About	&About...	mnuAbout	1
	Deselect the Enabled option.		

Note Notice that the menu names include only the prefix and the menu function. We shortened the names for this exercise to facilitate completing the walk through.

6. Choose OK.

7. Double-click the command button tool in the Toolbox *twice*.

   Add two command buttons and place them appropriately on the form.

8. Set the following properties for the command buttons.

Button	Name	Caption
Cardfile	cmdCardfile	&Cardfile
PaintBrush	cmdPaintBrush	&PaintBrush

9. From the Project Window, choose View Code.

   Make sure that STARTUP.FRM is in the foreground.

10. In the Object drop-down combo box, choose General Declarations.

11. Add the following code:

```
Const RED = &HFF&
Const WHITE = &HFFFFFF          Make sure you type six Fs.
Const BLUE = &HFF0000           Make sure you type four zeros.
```

Set the hexadecimal values for the colors that you want. You can find these colors and several others in CONSTANT.TXT.

Note There is a second method for completing the step above: Open CONSTANT.TXT, and copy and paste the appropriate lines into the General Declarations section of the form.

---

12. In the Object drop-down combo box, select cmdCardfile and locate the Click event template.

    When the user clicks this command button, use the **Shell** function to start Cardfile in a regular window. Add the following code:

```
x% = Shell("c:\windows\cardfile.exe", 1)
```

13. In the Object drop-down combo box, select cmdPaintBrush and locate the Click event procedure template.

    When the user clicks this command button, use the **Shell** function to start Paintbrush in a regular window. Add the following code;

```
y% = Shell("c:\windows\pbrush.exe", 1)
```

14. In the Object drop-down combo box, select mnuRed and locate the Click event procedure template. Add the following code:

```
frmStartup.BackColor = RED
mnuRed.Checked = True
mnuWhite.Checked = False
mnuBlue.Checked = False
```

15. In the Object list box, select mnuWhite and locate the Click event procedure template. Add the following code:

```
frmStartup.BackColor = WHITE
mnuRed.Checked = False
mnuWhite.Checked = True
mnuBlue.Checked = False
```

16. In the Object drop-down combo box, select mnuBlue and locate the Click event procedure template. Add the following code:

```
frmStartup.BackColor = BLUE
mnuRed.Checked = False
mnuWhite.Checked = False
mnuBlue.Checked = True
```

17. From the Run menu, choose Start and test your application.
18. From the Run menu, choose End.

    Close the application. You may notice that the Exit and About commands are not implemented yet. You will complete those functions in the next module.

19. Save the form as STARTUP.FRM in \WALKTHRU\SAMPLES.
20. Save the project as STARTUP.MAK in \WALKTHRU\SAMPLES.

Summary

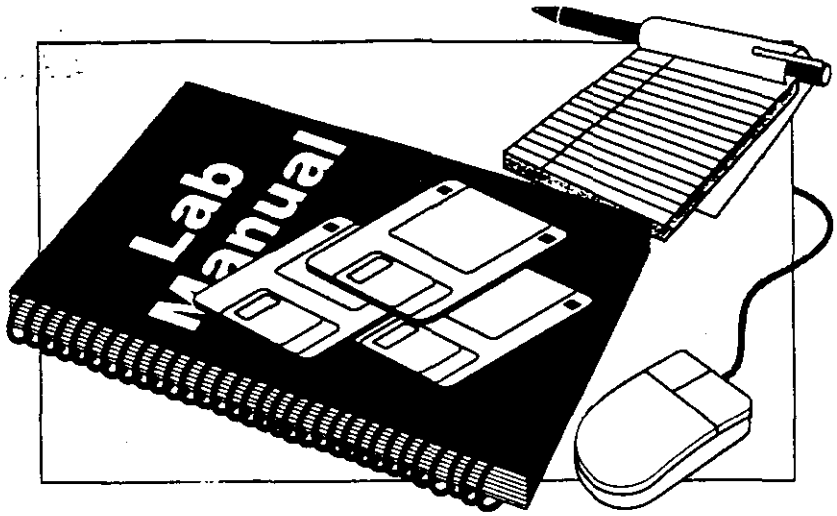
- **Menu Guidelines**
- **Microsoft Visual Basic Implementation**

Objectives

In this module you learned to:

- List and explain the use and value of all of the key menu elements:
  - Access or hot keys
  - Shortcut keys
  - Menu bar
  - Separator bar
  - Ellipsis
- Create an application that uses at least one menu on an application window and uses more than one form. It should include a least one functionality menu as well as a Help menu that includes choices to display a simple Help form and an About box.

Lab Time



Go to the Creating Application Menus portion of your lab manual.

Module 5: Connecting Forms

Σ Overview

- **Form Management**
  - Statements
  - Methods
  - Event Procedures
- **Setting the Startup Form**

Overview

This module has two broad goals. First, it provides an introduction to the concepts of event procedures and methods as they relate to managing the forms of your application. Second, it reviews all the important events and methods, and therefore is a crucial lead-in to the controls modules, and the general discussion of functions and statements as they are implemented in Visual Basic.

This module also contains the first actual hands-on experience you will get in writing code that will be implemented by user actions with the application interface.

Prerequisites

Prior to starting this module, you should already be familiar with these terms:

- Controls
- Forms
- Properties

Overall Objective

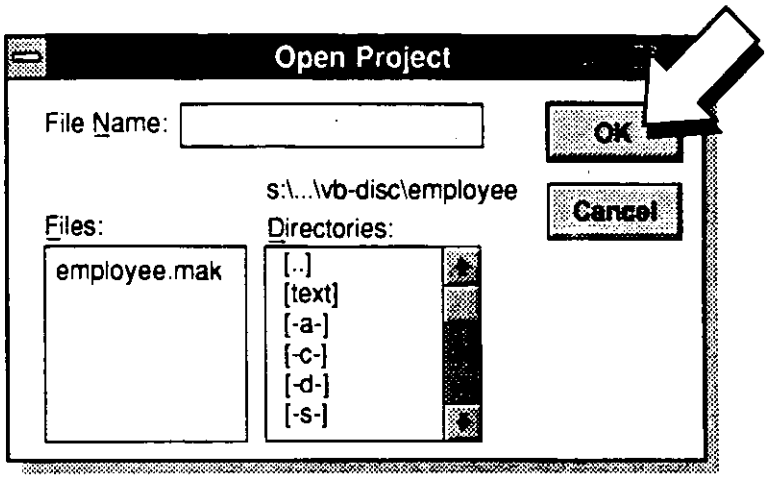
The primary objective of this module is to present the notion of event procedures and to discuss the essential events needed to manage forms.

Learning Objectives

At the end of the module, you will be able to:

- Use the **Unload** statement to unload a form from memory.
- Use the **Load** statement to load a form into memory.
- Use the **Show** method to display a form.
- Use the **Hide** method to hide a form but not unload it from memory.
- Set the startup form.

Statements



The Load Statement

The **Load** statement is used to load a form or control into memory. It normally appears within other event pr dures.

Syntax

Load *object*

Example

```
'Event procedure for a control on Form1
Sub Command1_Click ()
   Load Form2
End Sub
```

Important **Load** does not automatically display the form; it just loads it into memory. To make the form visible to the user, call the **Show** method, discussed later in this module. In some cases you may choose to preload all the forms in the application at application startup. Doing this causes them to be displayed more quickly when the **Show** method for the forms is called at the appropriate time. Of course this technique will add to the time it takes to start your application initially.

Further Examples

Application	Form	Procedure/Event
Calculator	CALC.FRM	Form_Load
		Cancel_Click

The Unload Statement

The **Unload** statement is used to remove a form or control from memory. Unloading a form from memory may be necessary when the memory resources used are needed for some other object or when you need to reset properties to their original value. Another typical use for the **Unload** statement is in an OK button Click event for a form like an About box. There is no reason to keep the About box form in memory, because it is seldom used.

Note that when a form is unloaded, only the display component is unloaded. The code associated with the form module remains in memory.

Syntax

Unload *object*

Example

```
Sub Command1_Click ()
   Unload Form1
End Sub
```

End Statement

This statement ends a Visual Basic program, procedure, or block. By itself, the **End** statement stops program execution, closes all files, clears the values of all variables, and destroys all forms.

You have seen or will see various uses for the **End** statement in other modules to end portions of code. The topic appears here because you can use the **End** keyword alone in code to end your application.

Syntax

End [{**Function** | **Select** | **Sub** | **Type** }]

Statement	Description
`End Function`	Ends a **Function** procedure definition
`End If`	Ends a block **If...Then** statement
`End Select`	Ends a **Select Case** block
`End Sub`	Ends a **Sub** procedure
`End Type`	Ends a user-defined type definition

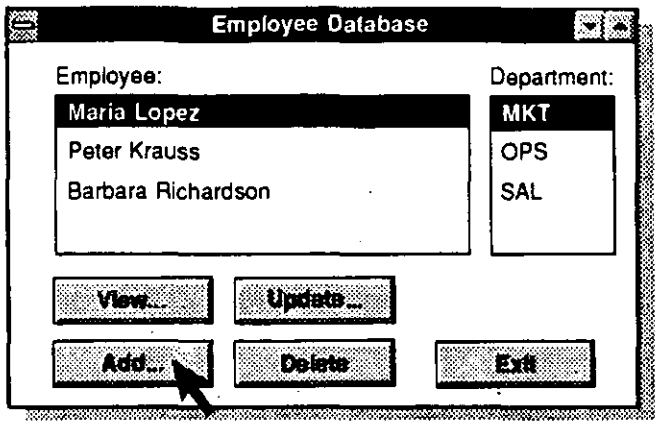
More examples are available in Visual Basic Help.

Example

```
'End Statement Example
Sub EndDemo ()
    Do                                      'Set up infinite loop
        Msg$ = "Enter A, B, C, or X."       'Enter X to exit
        UserInput$ = UCase$(InputBox$(Msg$))    'Get user input
        Select Case UserInput$                  'Evaluate input
            Case "A": Msg$ = "You entered 'A'."
            Case "B": Msg$ = "You entered 'B'."
            Case "C": Msg$ = "You entered 'C'."
            Case "X": End                   'End if user entered X
            Case Else: Msg$ = "You made an invalid choice.
                ^Try again."
        End Select
        MsgBox Msg$                         'Display results
    Loop
End Sub
```

Methods — Hide

EMPDB.HIDE



Hiding a Form

When a form is hidden, it is removed from the screen and its Visible property is set to **False**. A hidden form's controls are not accessible to the user, but they are available to the running Visual Basic application, to other applications communicating with the Visual Basic application through dynamic data exchange (DDE), and to timer events.

If a form is not loaded when the **Hide** method is invoked, the form is loaded but not shown.

Syntax

[form.] **Hide**

Note the optional form in the square brackets. Also note that the period within the brackets is not optional.

Example

The following example is available from Visual Basic Help.

```
1   'Hide Method Example
2   Sub Form_Click ()
3       Dim Msg$                          ' Declare variable
4       Hide                              ' Hide form
5       Msg$ = "Choose OK to make the form reappear."
6       MsgBox Msg$                       ' Display message
7       Show                              ' Show form again
8   End Sub
```

Further Examples

Application	Form	Procedure/Event
Menu	MAIN.FRM	mnuAppMgr_Click
		mnuEditor_Click
		mnuNum_Click
		mnuToDo_Click

Walk Through — Hide vs. Unload

Σ To see the difference between Hide and Unload

1. From the Walk Throughs program group, start Hide vs. Unload.

   When the application starts you see two forms. Form1, on the left, contains two buttons: one to show Form2, the other to unload Form2.

   Form2, on the right, contains nine buttons, Command1 through Command8, and a button labeled Hide Form2. The eight command buttons were placed on the form to make sure that Form2 uses enough memory and Windows resources to illustrate the point.

   The purpose of the application is to demonstrate the difference of the impact on memory and Windows-based resources when hiding a form versus unloading it.

2. Click the Hide Form2 button.

   This hides the form.

3. In Program Manager, choose About Program Manager from the Help menu.

   From the Help menu in Program Manager, choose About Program Manager. Note the values for the two lines at the bottom of the form for amount of Memory and System Resources free.

4. Choose OK.

   Close the About Program Manager dialog box.

5. Press CTRL+ESC to display the Windows Task List.

6. Double-click Hide.

7. Click the Unload Form2 button.

   Unload your Form2.

8. In Program Manager, choose About Program Manager from the Help menu.

   Note the values for the two lines at the bottom of the form for amount of Memory and System Resources free.

   Now that Form2 has been unloaded, not just hidden, you should see more memory and/or resources available. The amount will vary depending on your machine's environment.

9. Choose the OK button.

   Close the About Program Manager dialog box.

10. Press CTRL + ESC.

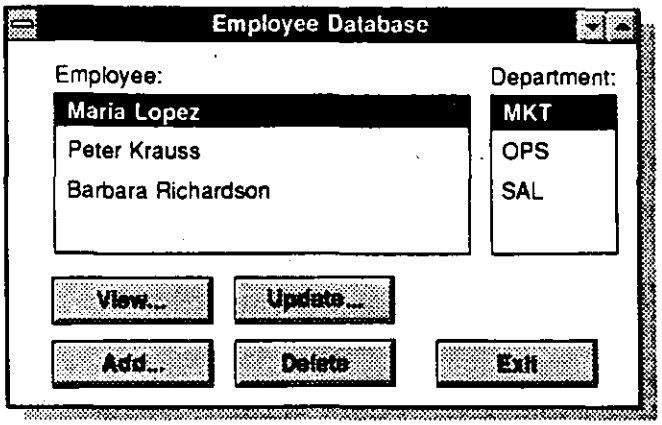
11. Select Hide from the Task List.

12. Click the End Task button.

    Terminate the project.

Methods — Show

EMPDB.SHOW



Showing a Form

Show is used to display forms that have been loaded. If you use the **Show** method on a form that has not been loaded, the **Show** method will automatically load the form and then show it.

Syntax

[*form.*] **Show** [*style%*]

Note The *style%* parameter is an integer value that determines whether the form is modal or modeless. If *style%* is 0, the form is modeless; if *style%* is 1, the form is modal.

Users must respond to a modal dialog box by clicking any one of a number of buttons. Any dialog box that lets the user continue working with other forms in the application is a modeless dialog box.

This example is available from Visual Basic Help.

Example

```
1   'Show Method Example
2   Sub Form_Click ()
3       Dim Msg$                                    ' Declare variable
4       Hide                                        ' Hide form
5       Msg$ = "Choose OK to make the form reappear."
6       MsgBox Msg$                                 ' Display message
7       Show                                        ' Show form again
8   End Sub
```

Further Examples

Application	Form	Procedure/Event
Hello	GETSTART.FRM	mnuButterfly_Click
		mnuHello_Click
		mnuScrollbar_Click
		mnuShell_Click

Walk Through — Connecting Forms in Startup

Σ To run the STARTUP.MAK project

1. If Visual Basic isn't running already, start it.

   A new, blank form should be on the screen.

2. From the File menu, choose Open Project.

3. Open STARTUP.MAK.

   This should be located in \WALKTHRU\SAMPLES. Load the code, but do not run the application yet.

4. From an MS-DOS prompt copy ABOUT.FRM from \STUDENT1\ABOUT to \WALKTHRU\SAMPLES.

5. From the File menu, choose Add File.

6. Choose ABOUT.FRM.

7. Choose OK.

   This adds ABOUT.FRM to the .MAK file for the Startup application.

8. Select STARTUP.FRM in the Project Window and choose View Code.

9. In the Object list box, select mnuAbout.

   Go to the Object list box and locate the template for the mnuAbout_Click event. It should be empty.

10. Add the following code:

```
frmAbout.Show 1
```

    Add the code that will show the About box when the user selects the Help menu and the About command. Remember the 1 causes this form to be modal. Notice the syntax here? FormName.Show.

Important You might be tempted to use the **Load** statement to display the frmAbout form, but you will find that won't work here. Loading a form only means that it is in memory. It doesn't mean that the form is actually displayed on screen. For that you need to use the **Show** method.

---

11. From the Run menu, choose Start.

    Start the application to test the result.

12. Open the Help menu and choose the About command on the Startup application.

    The About box from the lab appears on screen.

Note You will need to now change the Caption property on the About box form as well as the Icon and Text properties on the label to fit the current application.

---

13. Choose OK.

    Try to choose the OK button on the About Box form.

    You haven't coded that yet.

14. From the Run menu, choose End.
15. Select ABOUT.FRM.

    Select the ABOUT.FRM file in the Project window.
16. Choose View Code.

    Bring up the Code window.
17. In the Object list box, select cmdOK.

    Locate the code template for the cmdOK_Click event procedure.
18. Add the following code:

```
Unload frmAbout
```

    Add that code on the blank line in the template.

    Notice the syntax here? The **Unload** statement comes first, and the object name follows. Notice also that there is no punctuation.
19. From the Run menu, choose Start.

    Start the application again to test the success of your changes.
20. Open the Help menu and choose the About command on the Startup application

    This should display the About box.
21. Choose OK.

    Choose the OK command button on the About box, and it disappears from the screen.

    Now the question is what do you do to end or close the application itself?
22. From the Edit menu, choose Exit.

    Try exiting from this command.

    In order to close the application entirely, you still need to make one more change to the source code.

Note This placement of Exit is not standard for Windows. An application that follows interface guidelines for Windows would place Exit as the last item on the File menu.

---

23. From the Run menu, choose End.

    Stop the application.
24. Select frmStartup in the Project Window and choose View Code
25. In the Object list box, choose the Exit menu Click event procedure, and place the appropriate code there.
26. Add the following code:

```
End
```

    Put this code on the blank line in the template.

27. From the Run menu, choose Start.

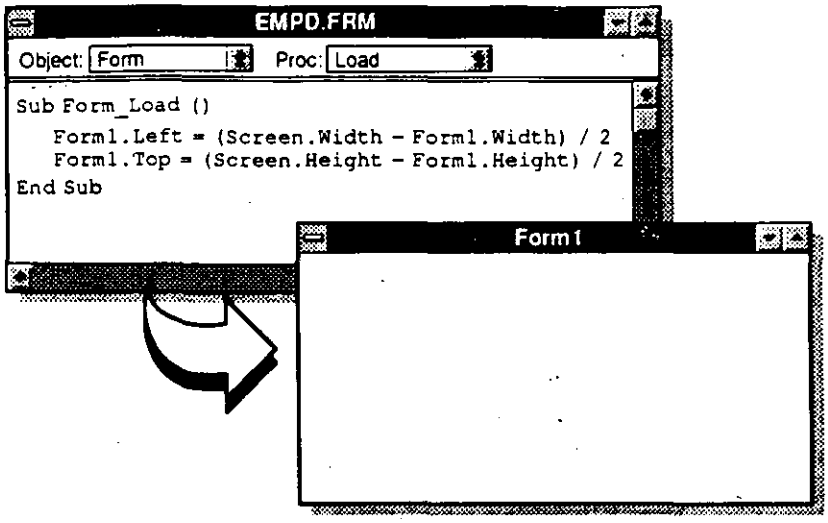
    Test the Exit command on the Edit menu, and test the About box elements.

28. From the Run menu, choose End.

29. From the File menu, choose Save Project.

    Save this project as STARTUP.MAK in \WALKTHRU\SAMPLES.

Event Procedures



Event Procedures

An event procedure is defined as a procedure invoked by a user or system-triggered event. Event procedures are always attached to a given form or control, and the syntax for an event procedure looks like the following:

Syntax

Sub *objectname_eventname*

statementblock

End Sub

Examples of event procedures are: Command1_Click and Form_Click.

Loading a Form

A Form Load event occurs when a form is loaded. It normally occurs when an application is run or as the result of either a **Load** statement or an implicit load that is caused by any reference to an unloaded form's properties or controls.

Typically you place initialization code for a form in a Form Load event procedure. For example, you would specify default settings for controls, initialize the contents of a combo or list box, and initialize form-level variables in a Form Load event.

Syntax

Sub Form_Load ()

    Initialization code

End Sub

Example

```
1   Sub Form_Load ()
2       ' Center the form on the screen
3       Form1.Left = (Screen.Width - Form1.Width) / 2
4       Form1.Top = (Screen.Height - Form1.Height) /2
5   End Sub
```

Unloading a Form

An Unload event happens when a form is about to be removed from memory. When that form is reloaded, the contents of all the controls are reinitialized. Normally, this event is triggered by a user action, by choosing Close from the Control menu or by an **Unload** statement.

Many times the Form_Unload event procedure contains a **Select Case** statement to verify that unloading should proceed or to specify actions that need to be taken prior to unloading.

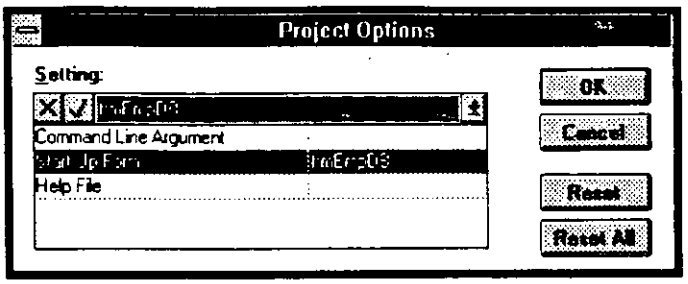
```
Sub Form_Unload (Cancel As Integer)

End Sub
```

Complete information on the Unload event is available from Visual Basic Help.

Further Examples

Application	Form	Procedure/Event
Menu	APPMGR.FRM	mnuDelApp_Click
		mnuExit_Click
	EDIT.FRM	mnuClose_Click
		mnuExit_Click
	NUMSYS.FRM	mnuExit_Click
	TDABOUT.FRM	Command1_Click
	TODO.FRM	mnuEdit_Click

Setting the Startup Form



Setting the Startup Form

If you have a multiple-form application, by default the first form you create will be the startup form. It is easy to make another form the startup form as you will see in the demonstration.

Walk Through — Setting the Startup Form Comment

To set the startup form

1. Start Visual Basic.
2. From the File menu, choose Open Project.

   Locate EMPLOYEE.MAK in \SOLUTION\FORMS.
3. EMPLOYEE.MAK.

   Load the Employee Database application into the Project window.
4. From the Options menu, choose Project.
5. Choose the Select Startup Form option, and press the drop-down arrow on the combo box at the top.
6. Select frmEmpDB from the list.
7. Choose OK.

   Set the startup form.
8. Save the project.
9. Quit Visu.. Basic.

Summary

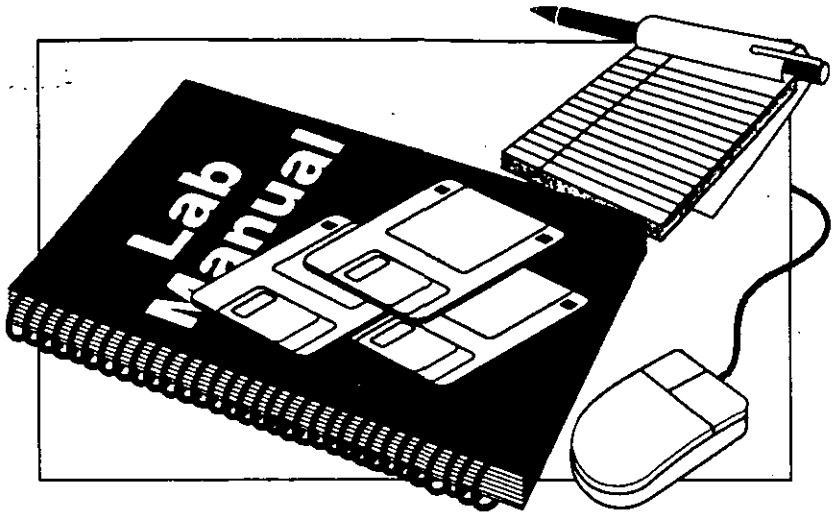
- **Form Management**
  - Statements
  - Methods
  - Event Procedures
- **Setting the Startup Form**

Objectives

In this module you learned to:

- Use the **Unload** statement to unload a form from memory.
- Use the **Load** statement to load a form into memory.
- Use the **Show** method to display a form.
- Use the **Hide** method to hide a form but not unload it from memory.
- Set the startup form.

Lab Time



Lab Time

Go to the Loading and Unloading Forms portion of your lab manual.

Module 6: Using Controls

Σ Overview

- Types of Controls
- Properties for Controls

Overview

The purpose of this module is to make sure that you understand the general purpose of control properties and events as well as the essential properties for each of the controls in the Toolbox.

This is not intended to be a comprehensive treatment of all the possible properties and events for all of the controls. In addition, this module covers only half of the total number of controls in Visual Basic. The module following this covers the other half.

The general approach here is to let you practice changing the properties for each control and actually see how the change affects the control.

Prerequisites

Prior to starting this module, you should have a fundamental awareness of:

- The functionality of Microsoft Visual Basic
- The general use for each of the controls in the Toolbox
- The general methods for setting properties on controls

Overall Objective

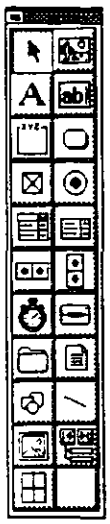
At the end of this module, you will be able to use key properties on half of the controls located in the Toolbox.

Learning Objectives

At the end of this module, you will have set key properties for:

- Labels
- Text boxes
- Frames
- Command buttons
- Check boxes
- Option buttons
- Combo boxes
- List boxes
- Horizontal and vertical scroll bars
- Timers
- Picture boxes

Types of Controls



This module covers the various clusters of controls found in the Toolbox, their most commonly used properties, and the events that are normally used with them. One of the first things you will see is that many of the controls have the same kinds of attributes. In many cases, a property will be covered only once, even though it shows up with almost every control.

Controls will be discussed in order of placement in the Toolbox, with the exception of the picture box, which will be covered last, and the drive, directory, and file list boxes, which will be discussed in a separate module.

A good way to get a handle on properties as they relate to controls is to look at the very front of the *Microsoft Visual Basic Language Reference*. There you can find a comprehensive list of controls and their properties.

Adding Controls to The ToolBox

The graphic on this page displays the default ToolBox. You can add controls to the Toolbox.

Σ To add a custom control to the Toolbox

1. From the File menu choose Add File.

   The Add File dialog is displayed.

2. From the Add File dialog, locate the custom control file that you want to add. For example, locate the file THREED.VBX. This file should be in the C:\WINDOWS\SYSTEM directory. This is a custom control provided by Visual Basic.

3. Select the file THREED.VBX and choose OK.

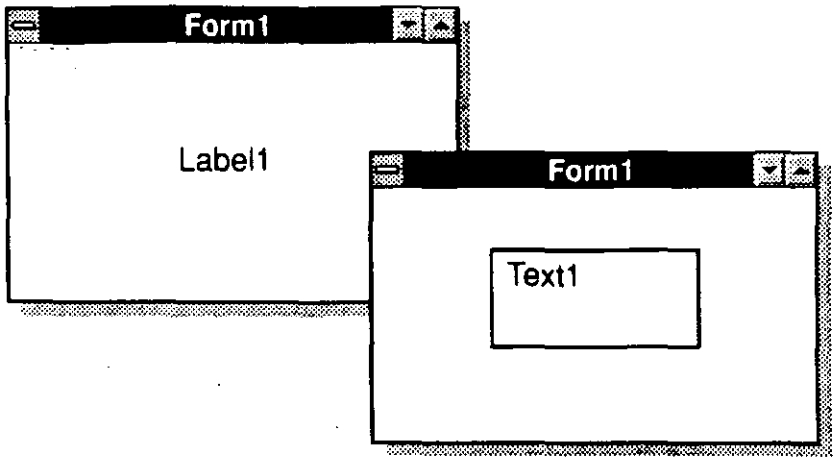
   The new controls are added to the Toolbox. You may get an error message if THREED.VBX has already been added to the Toolbox.

4. Use this same process to add any custom control to the Toolbox.

∑ Properties for Controls

- Labels, Text Boxes, and the Masked Edit Text Box
- Regular and 3-D Frames, Check Boxes, and Option Buttons
- Regular and 3-D Command Buttons
- Combo and List Boxes
- Horizontal and Vertical Scroll Bars
- Timers
- Picture Boxes

Labels and Text Boxes



Labels and Their Common Properties

Labels are most commonly used to display text that you don't want the user to be able to change. Typically this would be a caption under a graphic or captions for drive, directory, and file list boxes.

Property	Default	Comments
Alignment	0 - Left Justify	
AutoSize	**False**	Determines whether a control adjusts to fit its contents.
BackColor	White	
BorderStyle	0 - None	
Caption	Label1	Title for the control on screen. The caption is the default property for this control. Syntax—**Label1 = "Files"**
FontName	Helv	
FontSize	8.25	
ForeColor	Black	
Height	500	In twips.
Left	–	From left border of form.
Name	Label1	Name used in code.
		Use **lbl** as a prefix.
TabIndex	Assigned by Visual Basic	Determines the tab order of the label.
TabStop	True	Determines whether use can use Tab key to move to the text box.
Tag	–	User-defined value.
Top	–	From top border of form.

Text Boxes and Their Common Properties

A text box control displays either information that is provided by the application or information that the user can type.

Typical uses of text boxes would include the box where users type the string of characters they want the application to find in a search function. If you were creating a front-end for a database, you could use a sequence of text boxes to gather information about members of the database.

Property	Default	Comments
Alignment	0 - Left Justify	
FontName	Helv	
FontSize	8.25	
Height	500	In twips.
Left	–	From left border of form.
MaxLength	0	Controls amount of user input. Ranges from 0 to 64 characters.
Multiline	**False**	Means no multiline capability.
Name	Text1	Name used in code.
		Use **txt** as a prefix.
PasswordChar		Specifies the character to be displayed when user types text in the text box.
ScrollBars	0 - None	
Text	Text1	For none, assign "". This is the default property for the control.
		Syntax—**Text1** = ""
Top		From top border of form.
Width		Width of text box.

Walk Through — Text Box with a Scroll Bar

Σ To use the text box with a scroll bar

1. From the Walk Throughs program group, start Text Box with a Scroll Bar.

   When you start the application, you will see a form with a text box with a vertical scroll bar in the center of the form.

   The purpose of this application is to demonstrate the behavior of a text box with a scroll bar and how to create a text box with a scroll bar.

2. Type some text into the text box.

   Type enough text into the text box to cause it to start scrolling. Notice that the text will automatically wrap at the right margin. Notice also that you can select text and then cut and paste it just as in a standard Windows-based text box. Use CTRL+C to copy, CTRL+V to insert, CTRL+X to cut and CTRL+Z to undo.

3. Double-click the Control menu.

   Close the walk through.

4. Start Visual Basic.

5. Double-click the text box tool in the Toolbox.

   This will create a text box control.

6. In the Properties window drop-down combo box, make these changes:

   Height 1000 (approximately)

   ScrollBars 2 - Vertical

   Set the ScrollBars property to 2 - Vertical. Notice that you don't see the control change. There's a little problem here. Setting the ScrollBars property for a text box only becomes effective if the MultiLine property for the text box is also set to **True**.

7. Set the MultiLine property to **True**.

   Notice that a vertical scroll bar now appears on the text box.

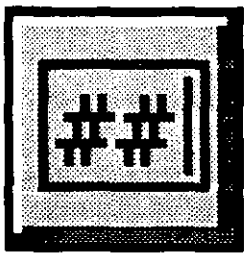
8. From the Run menu, choose Start.

   Test the behavior of the text box to see if it behaves as it did in the demo.

9. From the Run menu, choose End.

   Close the application.

Masked Edit Control



Masked Edit Control

This control is a special form of the text box control. It is used to restrict user input as well as to format data output.

Mask characters	Description
.	Decimal placeholder
,	Thousands separator
:	Time separator
/	Date separator
\	Treat next character as a literal
&	Character placeholder
A	Alphanumeric placeholder
?	Letter placeholder
#	Digit placeholder

Walk Through — Validating User Input

To monitor user input with the masked edit control

1. From the Walk Through program group, start Masked Edit.

   The purpose of this walk through is to demonstrate one of the routine uses of the masked edit control—displaying a text box that prompts for and monitors the correctness of a user-added telephone number.

2. Type a telephone number.

   Notice that the use of the parentheses and the underlining prompts the user on the type and amount of information to be typed. Notice also that the masked edit text box automatically tabs the cursor to the next control as you type the telephone number.

3. Close the application.

Σ To create the application

1. Start Visual Basic.
2. Resize the form and set its properties as follows:

   Height 2850 (approximately)

   Width 4560 (approximately)
3. Place a label control at the top of the form and set its properties as follows:

   Alignment 2 - Center

   AutoSize **True**

   Caption Customer's Telephone Number

4. Place the masked edit control in the center of the form and set its properties as follows:

   Mask (###) ###-####

   Width 1635 (approximately)

   Visual Basic calls the parentheses and the hyphen "literals." They are the visual cues that tell the user the type of data that is expected, and you type them as part of the input mask. The number signs (#) are called "digit placeholders," and they specify that a digit is required. As you have seen, when you run the application, the control places an appropriate number of underscores that tell the user the total number of character the number as well as the user's relative position within the number.
5. From the Run menu, choose Start.

   Test the behavior of the masked edit control to see if it behaves as it did in the demo.
6. From the Run menu, choose End.

ValidationError Event

If a user types an invalid character, a ValidationError event occurs. You can place code in the ValidationError event to display an error message. For example, you might want to beep if the user types an invalid character.

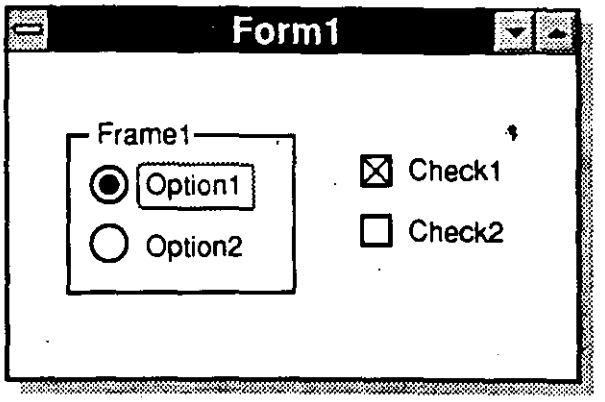
1. Place the following statement in the MaskedEdit1_ValidationError event:

```
Beep
```

2. From the Run menu choose Start. Type a letter in the masked edit field.

   The system beeps.
3. Save the form and .MAK file with appropriate names in \WALKTHRU\SAMPLES.

Regular and 3-D Frames, Check Boxes, and Option Buttons



The frame, check box, and option button controls provide a way to present users with a set of options from which they can choose.

For example, if you start the Solitaire game in Windows, then open the Game menu, and then choose Options, you will see a dialog box that contains two sets of option buttons. Inside the frame labeled Draw are the options affecting how many cards are drawn. Inside the other frame labeled Scoring are the options that affect how the game is scored. The arrangement on the screen and the use of frames tell the user and the system that these are independent groups of option buttons. A choice made in the Draw group will not affect the Scoring group, and vice versa.

Frames and Their Common Properties

Frames provide a means for graphically and functionally grouping controls. Frames are most commonly used to group two or more option buttons in a set of mutually exclusive choices.

Note Information presented in frames and option buttons can also be presented in drop-down list boxes.

Property	Default	Comments
Caption	Frame1	On screen label. This is the default property for this control. Syntax—**Frame = "Definitions"**
Name	Frame1	Name used in code. Use **fra** as a prefix.
Visible	**True**	**False** hides the frame and any controls inside it.

Caution When Placing Controls on Frames

There are two methods for placing controls on a form. The simplest method is to double-click a control icon. An instance of the control will appear in the center of the form, and you can position the control from there. This is a particularly useful approach when you are placing several controls on a single form.

The second approach is to single-click a tool in the Toolbox, move the cursor to the form, and drag the control on the form until it is the size you want. *This second method is required for placing controls on a frame.* If you do not use this method, you will lose the controls when you try to move the frame around on the form.

3-Dimensional Frames

Added property	Default	Comments
Alignment	0 - Left Justify	You cannot center- or right-justify frame captions with the other controls.
Caption		This is the default property for this control. Syntax—**Frame1 = "Definitions"**
Font3D	0 - None	Sets the amount and direction of shading in the Caption.
Name	Frame3D1	Name used in code. Use **fra** as prefix.

Check Boxes and Their Common Properties

Property	Default	Comments
Caption	Check1	On-screen value.
Enabled	**True**	**False** disables all user access.
Name	Check1	Name used in code.
		Use **chk** as a prefix.
Value	0 - Unchecked	1 - Checked; 2 - Grayed.
		(Fills box with gray). This is the default property for the control.
Visible	**True**	**False** hides the check box.

Events

Click Normally a Click event indicates that the user has made a selection. In the case of a check box, the Click event acts as a toggle. If there is an X inside the check box when the user clicks it, the X will be removed. If the check box is empty, an X will be placed inside the check box after the user clicks it. In either case, the Value property will be reset. A typical use of a check box might be to allow the user to choose between bold or normal font.

Option Buttons and Their Common Properties

Option buttons, as noted previously, are used to display an array of choices from which the user may select only one. An example is found in the Microsoft Windows Write Print Setup dialog box, where the user can select either the default printer or a specific printer and either portrait or landscape page orientation.

Property	Default	Comments
Caption	Option1	On-screen value.
Name	Option1	Name used in code. Use **opt** as a prefix.
Enabled	**True**	User has access to control.
Value	**False**	This is the default property for this control. Syntax—**Option1 = False**

Style Guidelines

Always have one option button in a group selected as the default for the user when the form is displayed.

Events

Click When the user clicks an option button, the Click event actually invokes several activities. It paints the dot on the selected option button, removes the dot from any other option button within the frame, and resets the appropriate values for all option buttons in the same group.

Walk Through — Option Buttons and Displaying Their Values

To compare the effect of the Click event for a check box versus an option button

1. From the Walk Throughs program group, start Check versus Option.

   When you start the application, you will see a form with three option buttons, two check boxes, and a clear screen button on the right side.

   The purpose of this application is to compare the effect of the Click event for a check box versus an option button.

2. Click the Option1 option button.

   The values for all three buttons should be displayed on the left side of the form.

3. Click the Option2 option button.

   The updated values for all three buttons should be displayed on the left side of the form.

4. Click the Check1 check box to place an X in it.

   The value of Check1.Value is printed on the form.

   Compare the number that is printed for the value of a check box when it is selected (1) to the value that is printed for an option button when it is selected (-1).

5. Click the Check2 check box to place an X in it.

   The value of Check2.Value is printed on the form.

   Compare the effect of the Click event for check boxes versus option buttons. Clicking one option button affects the state of the other option buttons in its group. Clicking a check box will not affect the state of other check boxes.

6. Double-click the Control menu.

   Close the walk through.

∑ To code values for option and check boxes

1. If Visual Basic is not running already, start it.
2. Double-click the frame control tool in the Toolbox.

   Add a frame to the form. Click and drag the sizing handles to make it big enough to hold three option buttons. Place it near the lower-right corner of the form so that there is plenty of room to display the values for the three option buttons you will add.

3. Click the option button tool in the Toolbox.
4. Move the mouse over the frame you just created on Form1.

   Note that the mouse pointer has changed from an arrow to a cross hair.

5. Click and drag inside the area of Frame1 to draw an option button there.
6. When the button is the size you want, release the mouse.
7. Draw two more buttons inside Frame1.
8. Double-click the top option button.

   This displays the FORM1.FRM Code window with the Option1_Click event procedure template.

9. Add the following code:

```
Print "Option1.Value = "; Option1.Value
Print "Option2.Value = "; Option2.Value
Print "Option3.Value = "; Option3.Value
```

   This adds the code that prints the values for the buttons.

10. In the Object list box, select Option2 and locate the Click event template from the Procedure list box.
11. Add the following code:

```
Print "Option1.Value = "; Option1.Value
Print "Option2.Value = "; Option2.Value
Print "Option3.Value = "; Option3.Value
```

    This adds the same code that prints the values for the buttons to the Click event for Option2.

12. Using the Copy command, place the code above into the Clipboard.
13. In the Object list box, select Option3 and locate the Click event template from the Procedure list box.

14. Using the Paste command, copy the contents of the Clipboard to this event procedure template.

    The code should look like this:

```
Print "Option1.Value = "; Option1.Value
Print "Option2.Value = "; Option2.Value
Print "Option3.Value = "; Option3.Value
```

    This adds the same code that prints the values for the buttons to the Click event for Option3.

15. From the Run menu, choose Start.

    Test your code.

16. From the Run menu, choose End.

    Close the application.

17. Quit Visual Basic.

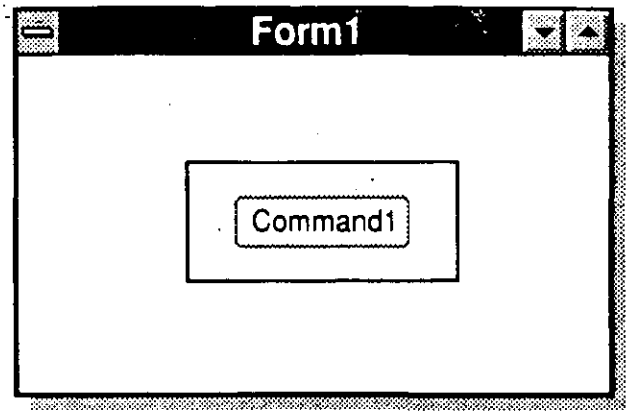
3-Dimensional Frames, Option Buttons, and Check Boxes

For the most part, these three controls are identical to the three controls just discussed. However, there are a few subtle differences that need to be noted as well as the procedures for setting the 3-D qualities for these controls.

3-Dimensional Option Buttons and Check Boxes

Added property	Default	Comments
Name (for check boxes)	Check3D1	Name used in code. Use **chk** as a prefix.
Name (for option buttons)	Option3D1	Name used in code. Use **opt** as a prefix.
Font3D	3 - Inset with Light Shading	Sets or returns the three-dimensional style of the caption. For example, you can set Font3D to have font appear raised or inset with light or dark shading.

Regular and 3-D Command Buttons



Command Buttons and Their Common Properties

A command button performs a task when the user presses it.

Property	Default	Comments
Cancel	**False**	**True**—button activated by ESC.
Caption	Command1	On-screen value.
Default	**False**	**True**—button activated by ENTER.
Enabled	**True**	**False** disables the button for users.
Height	500	
Left	–	From left border of form.
Name	Command1	Name used in code.
		Use **cmd** as a prefix.
Top	–	From top border of form.
Width	1215 twips	
Value		Syntax—**Command1 = Enabled**

Style Guidelines

Place buttons in a group on a form, either horizontally along the bottom of the form, or vertically along the right side of the form.

Events

Click With command buttons, a Click event normally means that the user wants some kind of action to take place. In the case of an OK button, for example, users might want a form removed from the screen so that they can continue work. In this case, you will be providing code.

When the user uses the mouse pointer to choose a command button, the Click event changes the color of the button (to simulate its being pressed) and executes whatever code you have added to the Click event procedure. A user can also use the keyboard to generate a Click event for a button by tabbing to the button and then pressing the SPACEBAR.

Example

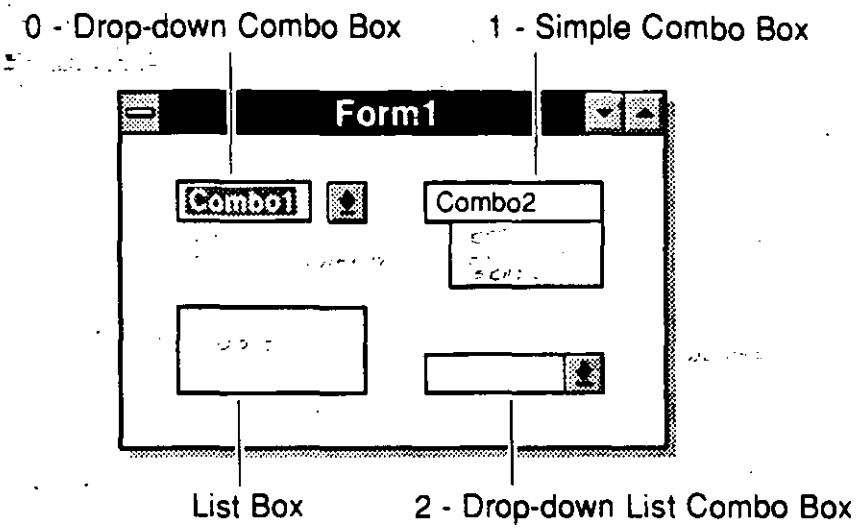
```
Sub Command1_Click ()
   ' Stop all execution of the program
   End
End Sub
```

3-Dimensional Command Buttons

This form of the command button behaves in much the same manner as the other command buttons, but there are a couple of interesting differences.

Property	Default	Comments
Bevel Width	2	Specifies the thickness of the highlight and shadow around the button. Valid range is from 0 to 10.
Name	Command3D1	Name used in code.
Outline	**True**	**False** disables the border or line edging the button.
Picture	None	Enables placing a picture or icon on a command button.

Combo and List Boxes



There are actually four kinds of list box controls, each of which has a slightly different functionality.

Both drop-down styles of boxes are designed to save on the initial amount of screen space in use by the control.

Users can only add new items directly into either a drop-down or simple combo box. With a list or drop-down list, they are limited to the choices provided by the application.

List Boxes and Their Common Properties

List boxes display a list of items from which the user can select only one at a time. Through the list box control, users can only select from the given list of articles, but you can write code that allows them to add and delete items during run time through some other control.

Automatic Scroll Bars

All list and combo boxes will automatically add a vertical scroll bar if the number of items contained in the list is greater than what will fit in the maximum display size of the control, which is set by the Height property.

Property	Default	Comments
Columns	Default	Enables multiple-column display of data in a single list box.
List		Accessible only at run time. Sets or returns the items contained in a control's list string array.
ListCount		Returns the number of items in the list.
ListIndex		Returns the index of the selected item in the list.

Property	Default	Comments
MutliSelect	0 - None	Other options: 1 - Simple and 2 - Extended. None means that the user can select one within the list. Simple means user can select multiple items using the cursor. Extended means user can press CTRL to select multiple items.
Name	List1	Name used in code. Use **lst** as a prefix.
Sorted	**False**	
Text		This is the default property for this control. Syntax—**List1 = "Item1"**

Events

Click With any of the list boxes, a Click event occurs when users select an item from the list to indicate their preferences.

DblClick A double click normally is used as a shortcut that combines two actions: the selection and the starting up of a process.

Methods

These are used for list management.

Name	Function	Syntax
AddItem	Used to add individual items to list box. Seeding a list box is done from Form_Load.	**List1.AddItem "*name*"**
RemoveItem	Used to remove items one at a time from a list box.	**List1.RemoveItem (ListCount)**
Clear	Used to remove all items in a list box in one stroke.	**List1.Clear**

Walk Through — Adding and Removing List Items

To add and remove list items

1. From the Walk Throughs program group, start List Box.

   When the application starts up, the cursor should already be placed in the text box.

   The purpose of this application is to demonstrate a simple procedure for adding items to and removing items from a list box.

2. Type **Harwood, Gene** in the text box.

   Add a new name in the text box.

3. Choose Add.

   You can do this with either the mouse or the ENTER key.

4. Select the text box.

   Tab to or click the text box.

5. Type **Simons, Anna** in the text box.

   Add a second name to the list box, and a vertical scroll bar should appear on your display.

6. Choose Pfeiffer, Terry.

   Single-click this list item.

7. Click the Remove button.

   The list item is removed from the display.

The question here is this: How does all of this work?

∑ To code list box handling

1. If Visual Basic isn't running already, start it.
2. From the File menu, choose New Project.

Note If you want to do all of the work for this walk through, go to the next step and start changing the properties for the form. If you want to start from a completed interface, you can also find a partially completed form called LIST.MAK in \WALKTHRU\SAMPLES. If you choose this option, skip down to step 12, the point where you start adding code to the Form_Load event.

---

3. Set the properties for the form as follows:

Caption	Consultants
Name	Form1
Height	3075 (approximately)
Width	3420 (approximately)

4. Double-click the list box tool in the Toolbox.

   Add a list box to the form.

5. Set the following properties:

Name	List1
Height	1005 (approximately)
Sorted	**False**
Width	2775 (approximately)

6. Double-click the text box tool in the Toolbox.

   Add a text box to the form.

7. Set the following properties:

Name	Text1
Height	555 (approximately)
TabIndex	1
Text	Blank
Width	2775 (approximately)

8. Double-click the command button tool in the Toolbox.

   Add a command button to the form.

9. Set the following properties to create an Add button.

Caption	Add
Name	Command1
Default	**True**
Height	555 (approximately)
Width	1215 (approximately)

10. Double-click the command button tool in the Toolbox.

    Add a second command button to the form.

11. Set the following properties to create a Remove button.

Caption	Remove
Name	Command2
Height	555 (approximately)
Width	1215 (approximately)

    Now that you have created the interface, you need to add the code to make it function.

12. In the Project window, choose the View Code button.

    Make sure that the Form is selected.

13. In the Object list box, select Form.

14. In the Procedure list box, select Load.

    Add the initialization code that loads the names into the list box when the form is loaded at run time.

```
' Fill the list box at run time
List1.AddItem "Morton, Joan"
List1.AddItem "White, Don"
List1.AddItem "Marshall, Lynn"
List1.AddItem "Pfeiffer, Terry"
```

15. In the Object list box, select Command1 and locate the Click event procedure in the procedure list box.

16. Add the following code:

```
List1.AddItem Text1.Text
```

    This code takes whatever is in the Text property of the text box and adds it to the list box.

17. From the Run menu, choose Start.

    Run your application.

18. Type **Harwood, Gene** in the text box.

    Type Gene Harwood's name in the text box and press ENTER or click Add. His name should be added to the list box but not deleted from the text box.

19. From the Run menu, choose End.

    In order to clear the text box, you need to add a second line to the Click event procedure here.

20. Add the following code to Command1_Click:

```
Text1.Text = ""
```

This takes care of the simple addition of items to a list box.

21. From the Run menu, choose Start.

Try adding Gene Harwood's name one more time. It should work just fine, but what about removing it?

22. From the Run menu, choose End.

Removing items from a list box is fairly straightforward.

23. In the Object list box, select Command2_Click.

24. Add the following code:

```
List1.RemoveItem List1.ListIndex
```

This code removes the item indicated by the value of List1.ListIndex. The **RemoveItem** method removes items one at a time. With small lists, this might be suitable. With larger lists, you would use the **Clear** method.

The above code will cause an error if no list item is selected. You can avoid this by modifying the code as follows:

```
If List1.ListIndex = -1 Then
    msg$ = "Please Select a Consultant"
    MsgBox msg$, 64, "List Selection"
Else
    List1.RemoveItem List1.ListIndex
End If
```

25. From the Run menu, choose Start.

Add and delete a couple of names to see that the code works.

26. From the Run menu, choose End.

Close the application.

How about sorting this list? Don't you have to add a lot of code to sort the items in the list? No, you don't! You simply set the Sorted property to **True** for List1, and all items inserted in the list will be sorted automatically.

27. Close the Code window.

28. From the Properties list box, select Sorted for the list box.

Click the List1 list box on the form.

Set the Sorted property to **True**.

29. From the Run menu, choose Start.

Note that the names in the list are now sorted. Add and delete a couple of names to see that the sorted order of the list is maintained.

30. From the Run menu, choose End.

Close the application.

31. Save your form and .MAK files with appropriate names in \WALKTHRU\SAMPLES. You can overwrite the file if it already exists.

Walk Through — Combo and List Box Differences

To add entries to combo and list boxes

1. From the Walk Throughs program group, start Combos.
2. Type **Fred** in the drop-down combo box.

   If you press the DOWN ARROW key, the list of names is displayed.
3. Press ENTER.

   The name Fred is added to the list. If you add several more names, a vertical scroll bar is also added to the list.
4. Type **Natasha** in the simple combo box.
5. Press ENTER.

   The new name will be added to this list. Notice here that the scroll bar is also added.
6. Select the name Larry in the drop-down list box.

   This name appears in the box at the top of the list, but users cannot add items to the list directly.
7. From the Control menu choose Close.

   Close the application.

Combo Boxes and Their Common Properties

Property	Default	Comments
Name	Combo1	Name used in code.
		Use **cbo** as a prefix.
Style	0 - Dropdown Combo	1 - Simple Combo
		2 - Drop-down List
Text	Combo1	
Height	300	Simple combo—The height displayed both at design time and run time.
		Drop-down—Set height property at design time (or run time), but full height will not be displayed until the user drops down the list at run time.
Text		This is the default property for the control.
		Syntax—**Combo1 = "Natasha"**

Events

Change A Change event indicates that the contents of a control have changed. For a combo box, a Change event occurs whenever you edit the text box portion of the combo box or when you assign a new value to the Text property from code.

Drop-Down and Simple Combo Boxes

Drop-down combo boxes are an alternative to option buttons within frames; that is, they are designed to display a limited number of logically arranged choices from which the user may select one. A classic example of a drop-down combo box is found in the Windows Control Panel. From the Settings menu, the user can choose Ports and then set the baud rate for the modem at any one of eight possible values, or type a value into the text box portion of the control. The new value typed can then be added to the list. The drop-down combo box enables a scrollable list that allows the user to review more choices than can be easily presented on screen. The user's choice is displayed in the top box.

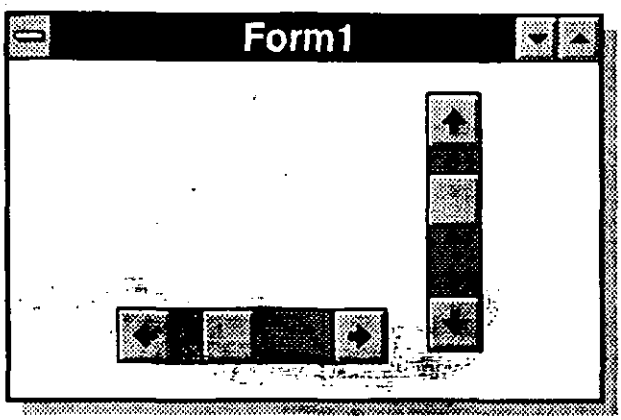
Drop-Down List Combo Boxes

Drop-down list combo boxes are used to present users with a limited number of choices, from which they may select only one.

Key Feature

Both drop-down combo and drop-down list combo boxes display data from which the user may select only *one* choice. But with a drop-down list users can only select from the list; they cannot add to it.

Scroll Bars



Scroll Bars and Their Common Properties

Scroll bars are most commonly used to enable quickly moving across a long list of items, such as a long list of filenames. They can also be used for indicating the current position on a scale, such as shades of coloring for customizing the look and feel of the screen. Or, scroll bars might be used to indicate the volume of an audio system.

Property	Default	Comments
LargeChange	1 (1–32,767)	Amount of change when user clicks scroll bar shaft.
Max	32,767	Maximum value of scroll bar handle in bottom most position.
Min	0	Minimum value of scroll bar handle in topmost position.
Name	HScroll1	Name used in code.
	–or–	Use **hsb** as a prefix.
	Vscroll1	Name used in code.
		Use **vsb** as a prefix.
SmallChange	1 (1–32,767)	Amount of change when user clicks a scroll arrow.
Value	0	This is the default property for this control. Can be set by user's direct interaction with the control or through code in response to other events.

Events

Change Occurs when the contents of the control have changed.

Walk Through — Scroll Bars, Properties, and the Change Event

To see scroll bars, properties, and the Change event

1. From the Walk Throughs program group, start Scroll Bars.

   You will see a form with a vertical scroll bar on it.

   The purpose of this application is to demonstrate the behavior of a scroll bar and the effects of the properties LargeChange and SmallChange on the Value property.

2. Click the arrows on both ends of the scroll bar.

   Note the change in the Value property.

3. Click on the shaft of the scroll bar both above and below the scroll box.

   Note the change in the Value property.

4. Close the application.

5. Start Visual Basic.

6. From the File menu, choose New Project.

   Start a new project.

7. Set the following form properties in the Properties window:

Height	4425 (approximately)
Width	5430 (approximately)

8. Double-click the vertical scroll bar tool in the Toolbox.

   Add a vertical scroll bar and change some of its properties:

Height	3495 (approximately)
LargeChange	20
Max	100
SmallChange	10
Width	375 (approximately)

9. In the Project window, choose the View Code button.

   Make sure that Form1 has been selected.

10. In the Object list box, select VScroll_Change.

    Locate the Change event procedure for the vertical scroll bar.

11. Add the following code:

```
Print "VScroll1.Value = "; VScroll1.Value
```

    Display the Value property each time it changes.

12. From the Run menu, choose Start.

    Test the application you have written.

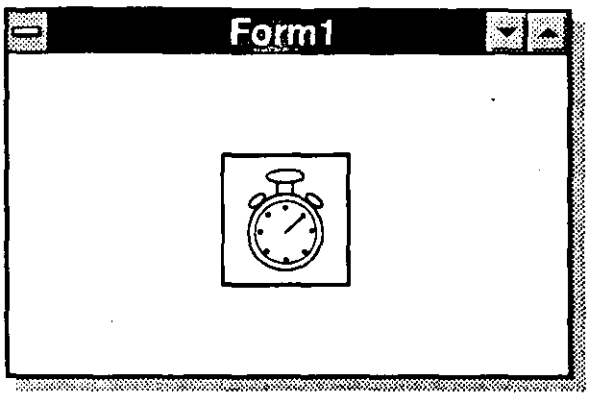
13. From the Run menu, choose End.

Close the application.

14. Quit Visual Basic.

15. Save your files and MAK files with appropriate names in C:\VBTHRUSAMPLES.

Timers



Timers and Their Common Properties

Timers are used to run events at a specific time or within an interval that you specify. For example, within a scheduling application, timers would be used to enable alarms for the user.

A special characteristic of this control is that it is never visible at run time.

Property	Default	Comments
Enabled	**True**	This is the default property for this control.
		Syntax—**Timer1 = Enabled**
Interval	0	Countdown interval for the timer, measured in milliseconds.
Name	Timer1	Name used in code.
		Use **tmr** as a prefix.

Walk Through — Clock Application and the Timer Control

To use the timer control

1. From the Walk Throughs program group, start Clock (Timer).

   When you start the application, you will see a form containing a digital clock displaying the date and time. The clock will be updated every second.

   The purpose of this application is to demonstrate the behavior and use of a timer control.

2. Double-click the Control menu.

   Close the walk through.

3. Start Visual Basic.

4. From the File menu, choose New Project.

   Start a new project.

5. Set the properties for the form as follows:

BackColor	Yellow
BorderStyle	3 - Fixed Double Also removes Min/Max buttons
Caption	Clock
Height	2235 (approximately)
Name	frmClock
Width	4345 (approximately)

6. Double-click the label tool in the Toolbox.

   Add a label to the form.

7. Set the properties for the label as follows:

Alignment	2 - Center
BackColor	Red
Caption	(none)
FontSize	12
ForeColor	White
Height	1340 (approximately)
Left	120
Name	lblDateTime
Top	420
Width	4015 (approximately)

8. Double-click the timer tool in the Toolbox.

9. Set the following properties:

Name	tmrTimer1
Interval	1000

   Place a timer on the form. Set the properties for it. You can place it anywhere. Behind the label will be fine, because timers are not displayed on screen at run time.

10. In the Project window, choose the View Code window.

    Make sure that frmClock has the focus.

11. In the Object and Procedure list boxes, locate the tmrTimer1_Timer.

    Locate the Timer event in the code.

12. Add the following code:

```
lblDateTime.Caption = Format$(Now, "mmmm d, yyyy h:mm:ss am/pm")
```

    Add the code on the blank line in the Click event template.

13. Double-click the Control menu.

    Close the Code window.

14. From the Run menu, choose Start.

    Test the clock application that you have just created.

15. From the Run menu, choose End.

    Close the clock.

16. Quit Visual Basic.

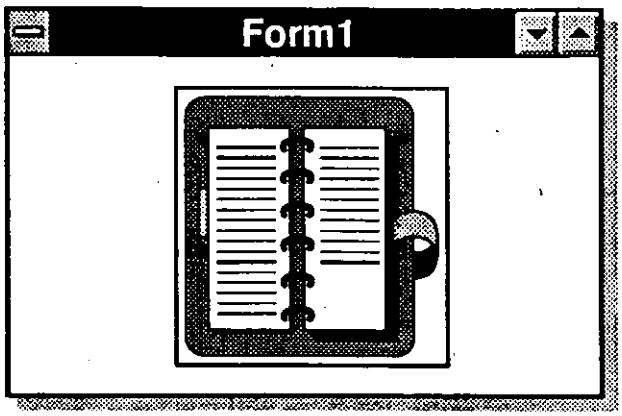
17. Save your form and .MAK files with appropriate names in \WALKTHRU\SAMPLES.

Using a Timer to Initiate a Task at a Specific Interval

You can use the timer to initiate a task after a specific interval. In the following example, the timer event starts a backup procedure. The timer event occurs after at least 60 seconds have elapsed. You could place code in the Timer event to check the current time and start the backup routine only at certain times.

```
1 Sub Form_Load ()
2     Rem Timer will go off after 60 seconds.
3     timer1.Interval = 60000
4     timer1.Enabled = True
5 End Sub

6   Sub Timer1_timer ()
7       x = Shell("c:\myb.  .p.bat")
8   End Sub
```

Picture Boxes



Picture Boxes and Their Common Properties

Picture boxes are used to display a range of graphics, from icons to bitmaps to metafiles. You could use a picture box, for example, to display the icons in the Visual Basic icons library. You could also use the picture control to display the kinds of card backs available for users of a Solitaire game.

Property	Default	Comments
AutoRedraw	**False**	Will not automatically repaint.
AutoSize	**False**	Will not automatically resize control to fit its contents.
Name	Picture1	Name used in code.
		Use **pic** as prefix.
Picture	(none)	Use Load Picture dialog box at design time to select a .BMP, .WMF, or .ICO file. This is the default property for this control.
		Syntax—**Picture1 = LoadPicture("*path*")**

Events

Paint This event occurs when a form or control is moved, exposing parts that weren't initially painted on screen. Note that this applies only when AutoRedraw is set to **False**. If AutoRedraw is set to **True**, repainting is done automatically, and no Paint event occurs.

Loading a Picture at Run Time

To load a new picture file from disk into a picture box at run time, call the **LoadPicture** function with the name of the file you want to load.

Example

```
Sub Command1_Click ()
   Picture1.Picture = LoadPicture ("\VB\ICONS\ARROWS\POINT12.ICO")
End Sub
```

If you want to copy a picture from one picture box to another at run time, use the Image property of the picture you wish to copy.

Example

```
Sub cmdCopyPicture ()
   Picture2.Picture = Picture1.Image
End Sub
```

Clearing a Picture Box

At design time, clear the Picture property by clicking the picture box you wish to clear. Make sure the Picture property is selected in the Properties list, click the settings box, and then press the DEL key. The settings box will now read (none).

At run time, clear a picture box's Picture property by calling the **LoadPicture** function with an empty argument.

Example

```
Sub Command2_Click ()
   Picture1.Picture = LoadPicture ()
End Sub
```

Walk Through — Picture Box Controls and AutoSizing

Σ To use picture box controls and autosizing

1. Start Visual Basic.
2. From the File menu, choose New Project.

   Open a new blank form.
3. Double-click the picture box icon in the Toolbox.

   Place a picture control on the form.
4. Set the following properties for the control:

AutoSize	**True**
Height	1000 (approximately)
Width	1000 (approximately)
5. Place an icon in the picture control.
6. Set the Picture property to:

   \VB\ICONS\ARROWS\ARW01DN.ICO

   The picture box automatically resizes to fit the graphic when it is added.
7. Place a bitmap in the picture control.
8. Set the Picture property to:

   \VB\BITMAPS\GAUGE\CIRCLOCK.BMP

   The picture box automatically resizes to fit this graphic.
9. To remove the Picture property:
   Click the Name property and then click back on the Picture property.
   Click in the Settings combo box for the Picture property.
   Press the DEL key

   The settings box will read (none). This will remove the graphic from the Picture property of the picture box.
10. Quit Visual Basic.

Summary

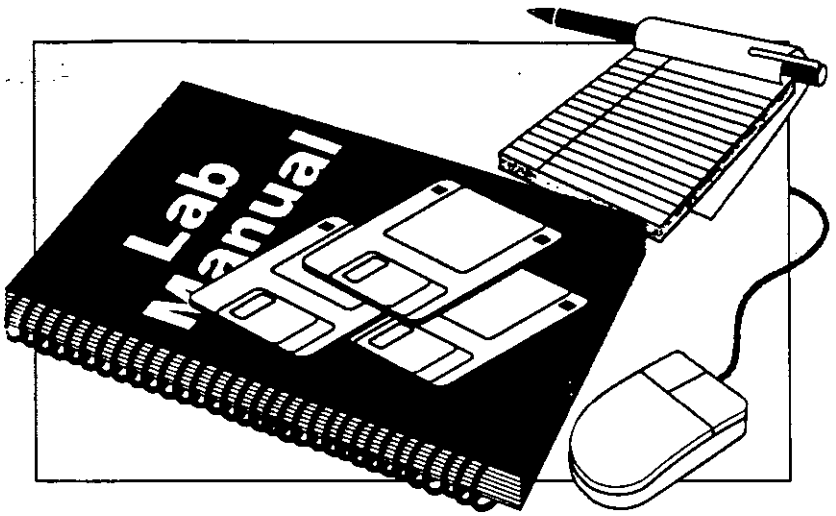
- Types of Controls
- Properties for Controls

Objectives

In this module you set key properties for:

- Labels
- Text boxes
- Frames
- Command buttons
- Check boxes
- Option buttons
- Combo boxes
- List boxes
- Horizontal and vertical scroll bars
- Timers
- Picture boxes

Lab Time



Go to the Adding Controls to Forms portion of your lab manual.

Module 7: File Browsers and Other Controls

Σ Overview

- File Browsing
- Grid Control
- 3-D Panel Controls and Group Push Button

Overview

This module is designed to directly follow the controls module, and it covers most of the rest of the tools in the Toolbox. However, because of time limits it does not cover picture clip, lightweight line and shape, image, and serial communications controls.

The first part of the module discusses file browsing controls—file, directory, and drive list boxes as well as the common dialog control. The first three are specialized controls that act just like regular list boxes; but as you will see, they have special features that make them distinct.

The purpose of this module is to walk you through the flow of events that are needed to connect the various controls. The emphasis here is on talking about the kinds of events that need to be used in order for the changes on one control to be reflected on the other control.

The rest of the module provides you with discussion of and walk throughs for most of the rest of the tools in the Toolbox.

Prerequisites

This module assumes prior experience with:

- List boxes
- Click event implementation

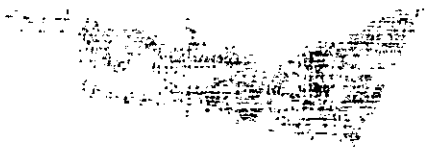
Overall Objective

The overall objective is for you to develop an awareness of the rest of the tools in the Toolbox.

Learning Objectives

At the end of this module, you will be able to:

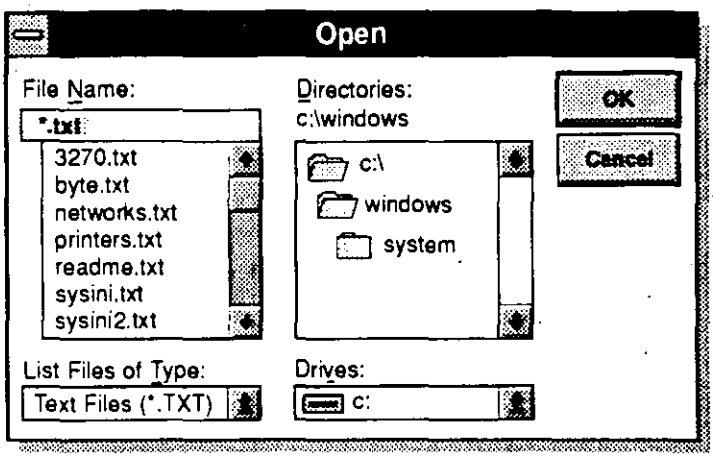
- Design and develop a form that lets users browse drives, directories, and files.
- Use the Change and PathChange events.
- Use the grid control.
- Use the group push button and 3-D panel controls.



Σ File Browsing

- Scenario
- Properties and Events for File Browser Controls
- The Common Dialog Control

Scenario



File Browsers

A quick review of the workings of the file browser in Notepad shows that a number of events occur when the user selects a different drive or directory. If the user selects a new drive, the directories and files list boxes are automatically updated, showing the current working directory for the system and any files (of the file type specified in the List Files Of Type list box).

Walk Through — How File Browsers Work

To use file browsers

1. In the Accessories group window, open Notepad.

   All of you have used a file browser of one sort or another, but you may not have looked closely at how it works.

2. From the File menu, choose Open.

   Drives list box The Drives list box is a special form of drop-down list box. The user can see options presented in the list but cannot add items to it.

   The more important element to notice at this point, however, is that when the user selects a new drive, the change in selection is reflected in the directory and file list boxes as well as in the current working directory.

   Directories list box The Directories list box is a special form of list box. It behaves like a regular list box but displays only directories and subdirectories. Users cannot add items to or delete items from the list.

   File Name combo box The File Name combo box is a special form of the list box that sends and receives messages from a text box. The text box allows users to add items to the list, something that cannot be done with a list box.

   List Files of Type Combo Box Notice that this, too, is a drop-down list box. Users can select items from it, but they cannot add items to it themselves.

   Current working directory Finally, notice that there is a label that tells the user the current drive and directory.

3. Choose Cancel, and close Notepad.

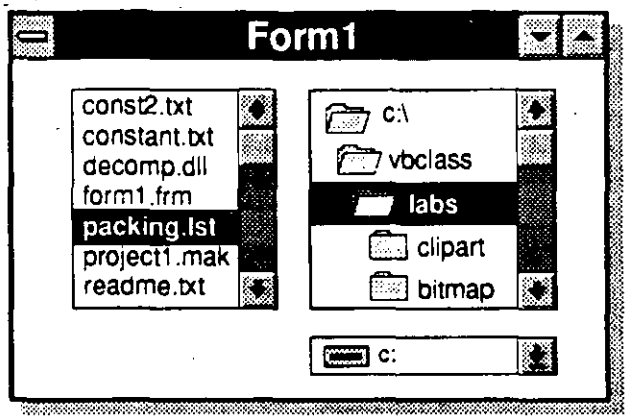
   End walk through.

Style Guidelines

Typically, file browsers can be set to display only a given type of file in the specified drive and directory. Because Notepad is designed to read files that have the .TXT extension, .TXT is the default file type and *.* is the second choice.

Because the Employee Database application is designed to locate .BMP and .WMF files, they will be the default file types.

Properties and Events for File Browser Controls



Drive, Directory, and File List Boxes

All three of these controls are special forms of list boxes. They have many of the same characteristics, but there are also a number of special properties that they use to enable navigating through a file system.

Drive List Boxes and Their Common Properties

The first of these controls is the drive list box. It is a special form of the drop-down list box. Initially it only shows the current drive for the form. However, if the user clicks the drop-down arrow, the control will display all of the other drives currently available on the system.

Property	Default	Comments
Name	Drive1	Name used in code.
		Use drv as a prefix.
Drive		Accessible only at run time, this property returns the selected drive. This is the default property for this control.

Synchronizing the Drive Property

Example

```
Sub Drive1_Change ()
   'When drive is changed, reset the directory list box path
   Dir1.Path = Drive1.Drive
End Sub
```

Directory List Boxes and Their Common Properties

Property	Default	Comments
Name	Dir1	Name used in code.
		Use dir as a prefix.
Path		Accessible only at run time, this property returns the absolute path for the selected directory. This is the default property for this control.

Synchronizing the Path Property

Example

```
Sub Dir1_Change ()
   ' When directory is changed, reset the file list box path
   File1.Path = Dir1.Path
End Sub
```

File List Boxes and Their Common Properties

Property	Default	Comments
Archive	True	Displays all files with archive bit on.
Name	File1	Name used in code.
		Use fil as a prefix.
FileName		Sets or returns the selected file from the list portion of a file list box. Accessible only at run time. This is the default property for this control.
Hidden	False	Does not display hidden files.
List		Accessible only at run time; returns the items contained in a list box.
ListCount		Accessible only at run time; returns the total number of items in the list box.
ListIndex		Accessible only at run time; returns the index of the currently selected item.
Normal	True	Displays all files with archive bit set on.
Path		Accessible only at run time.
Pattern	*.*	Display all files.
ReadOnly	True	
System	False	

Accessing the FileName Property

Example

```
Sub File1_DblClick ()
   Label1.Caption = File1.FileName
End Sub
```

Events

Click For all three of the basic controls, a Click event indicates that the user has made a selection. That is, the user is selecting or changing drives, directories, subdirectories, or filenames.

Double-click The only time that a double click is routinely used in a file browser is when the user uses the File Name combo box. In this case, the double click indicates that the user has located a specific file of interest and wants the application that uses the file browser to actually open that file.

Change The real workhorses of a file browser are the Change events. Code that you add to the event procedure template keeps the three controls synchronized whenever the user selects a new item in one of the list boxes.

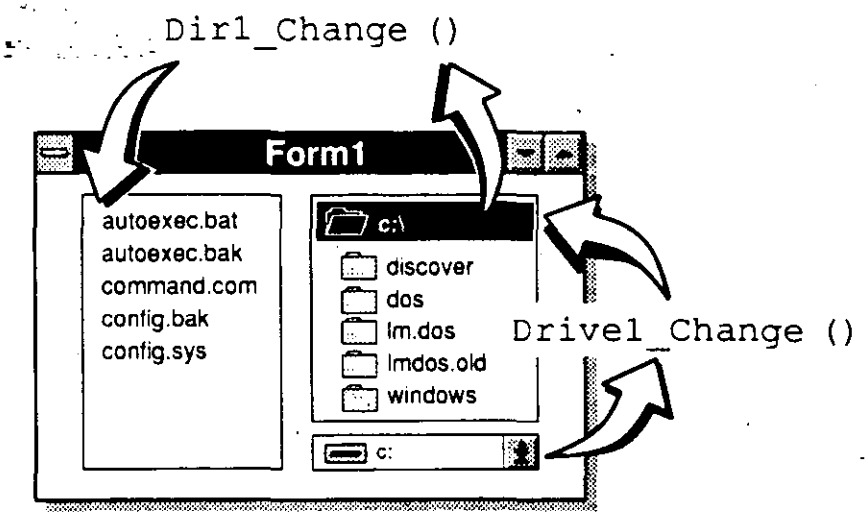
PathChange This event occurs when the selected path has been changed by setting the FileName or Path property from code.

PatternChange This event occurs when the pattern has been changed by setting the FileName or Pattern property from code.

Σ Change Events

- Change Event
- PathChange Event

The Change Event



Changing the Path for Differing List Boxes

Change Events Change events indicate that the contents of a control have changed. Change events are associated with a wide variety of controls including combo boxes, horizontal and vertical scroll bars, labels, and picture boxes. For our purposes, this module focuses on the Change events for directory and drive list boxes only.

Directory List Box The Change event is invoked when the user clicks a new directory or when the Path property is changed from the code. It indicates that the contents of the control have been changed.

Example

```
Sub Dir1_Change ()
   ' When directory is changed, reset the file path
   File1.Path = Dir1.Path
End Sub
```

Drive List Box The Change event is invoked when the user selects a new drive or when the Drive property is changed from the code. It resets the path for the directory list box.

Example

```
Sub Drive1_Change ()
   ' When drive is changed, reset the directory's path
   Dir1.Path = Drive1.Drive
End Sub
```

Walk Through — Drawing and Coding a File Browser

To draw and code the file browser

1. From the Walk Throughs program group, start Browser.

   When this application starts, you will see a simplified form of the file browser used in the Employee Database. This form has file, directory, and drive list boxes only.

   Click a new directory and notice the file list is updated.

   The purpose of the application is to demonstrate the Path and Drive properties and the Change events that are used to synchronize controls.

2. Close Browser.
3. Start Visual Basic.
4. From the File, choose New Project.
5. For Form1, set the following properties in the Properties window:

Height	3930 (approximately)
Width	4290 (approximately)

6. Double-click the file list box tool in the Toolbox.
7. Set the following properties:

Height	2760 (approximately)
Width	1455 (approximately)

8. Double-click the directory list box tool in the Toolbox.
9. Set the following properties:

Height	2175 (approximately)
Width	1455 (approximately)

10. Double-click the drive list box tool in the Toolbox.
11. Set the following properties:

Width	1455 (approximately)

    The first step of implementing a file browser is to add the Change event code that synchronizes changes in the state of the drive and directory list boxes. To do this, you need to:

12. Double-click the drive list box.

    This displays the Code window with the Drive1_Change event procedure in it.

13. Add the following code:

```
Dir1.Path = Drive1.Drive
```

    This code resets the Path property of the directory list box to that of the Drive property of the drive list box.

    Now the directory list box will be kept current with changes in the drive list box. But what about the file list box?

14. From the Object list, select Dir1.

    The Code window with the Dir1_Change event template is displayed.

15. Add the following code:

```
File1.Path = Dir1.Path
```

16. Click Form1 to display the interface.

    The code added above resets the Path property of the file list box to that of the Path property of the directory list box.

    The characteristics of the Path property of a directory list box vary depending on whether the user's current working directory is the root directory.

    If the user selects the root directory, the path will be "\"—that is, a path (in this case the root), which is a backslash. If the user selects any other directory, the path will end in the current working directory, with no backslash at the end.

    This is important when a program needs to build a fully qualified filename. The code must check to see if a backslash must be added to the current value of the Path property before concatenating it to a filename.

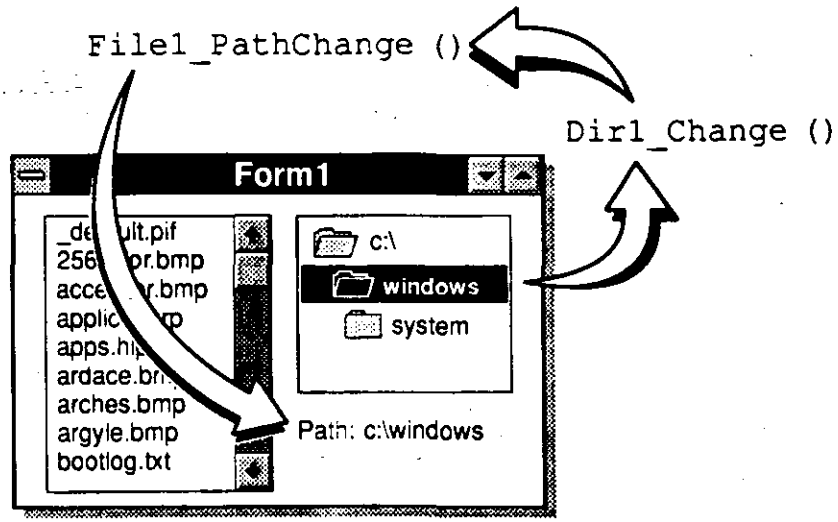
    The code to handle this requires a knowledge of conditional processing. To see how this is done, inspect the cmd OK_click procedure in the AddPhoto.FRM code in \SOLUTION\PRINTING.

17. From the Run menu, choose Start.

    Test to see if your code works.

18. From the File menu, choose Save As.
19. Save the file as BROWSER.FRM in \WALKTHRU\BROWSER.
20. Save the project as BROWSER.MAK in \WALKTHRU\BROWSER.

PathChange



PathChange

The event occurs when the selected path changes by setting either the FileName or Path property from code.

These two examples are from Visual Basic Help.

Examples

```
Sub Dir1_Change ()
   File1.Path = Dir1.Path
End Sub

'PathChange Event Example
Sub File1_PathChange ()
   Label1.Caption = "Path: " + Dir1.Path
End Sub
```

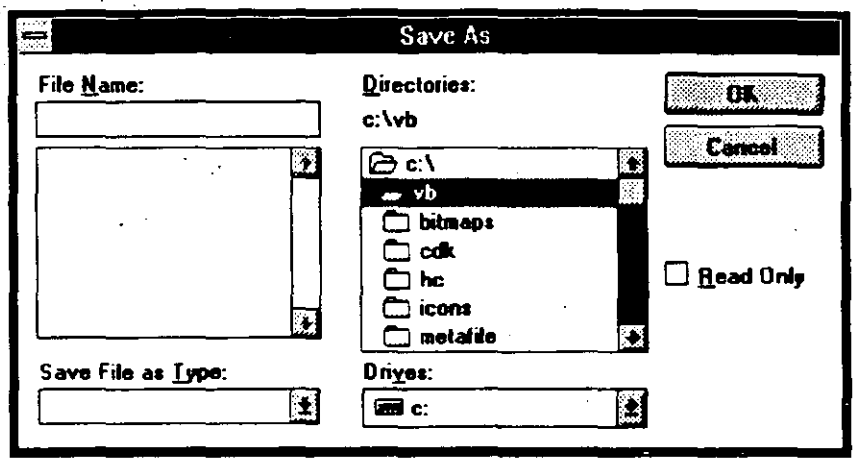
Notice in this example that the PathChange event is generated by code in a Change event in the directory list box control to synchronize the value of the File1.Path property to the new value for Dir1.Path.

For further examples using the PathChange event, see the following.

Further Examples

Application	Form	Procedure
IconWorks	VIEWICON.FRM	File_FileList_PathChange

The Common Dialog Control



The Common Dialog Control

The common dialog control is a versatile tool provided as a standard part of the Professional Edition of Visual Basic. The most common use for it is as a file browser, but as you will see, it has many other possible uses.

Example

```
Sub mnuFileOpen_Click ()
   CMDialog1.Action = 2
End Sub
```

Below is a list of the Action property settings and their associated dialog box type.

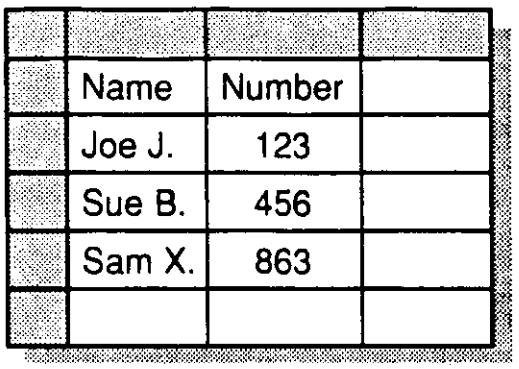
Action setting	Type of dialog box
0	No action
1	File open dialog box
2	File save dialog box
3	Color dialog box
4	Choose font dialog box
5	Printer dialog box
6	Invoke WINHELP.EXE

Σ More Controls

- Grid Control
- 3-D Panel Controls and Group Push Button

This final portion of the module deals, briefly, with each of the remaining tools in the Toolbox.

Using the Grid Control to Display Output



	Name	Number	
	Joe J.	123	
	Sue B.	456	
	Sam X.	863	

```
Grid1.Col = 1
Grid1.Row =1
Grid1.Text = "Name"
```

The Grid control allows you to display output on a grid.

The Cols and Rows properties set the total number of columns and rows you want in the grid. These properties are available at design time or at run time.

The Col and Row properties set or return the current cell in the grid. These properties are available at run time only. The Text property sets or returns the text in the current cell.

```
Sub cmdAddText_Click ()
   'Set the current cell; remember that the first row and
   'col are 0
   Grid1.Col = 1
   Grid1.Row = 1
   Grid1.Text = "Name"
End Sub
```

The Clip property sets or returns the contents of the grid's selected region—for example:

```
Grid1.Clip = "ABC"
```

You can include tabs and carriage returns in the string expression to indicate new columns and new rows, respectively.

The following example places text in a two-by-two range of cells.

Example

```
1   Sub cmdClipText_Click ()
2       Grid1.SelStartCol = 1
3       Grid1.SelEndCol = 2
4       Grid1.SelStartRow = 2
5       Grid1.SelEndRow = 3
6       tb$ = Chr$(9)                    'Tab character
7       cr$ = Chr$(13)                   'Carriage return
8       d$ = "Joe J" + tb$ + "123" + cr$ + "Sue B" + tb$ + "456"
9       Grid1.Clip = d$
10  End Sub
```

The grid control is used to display output. However, you can simulate input on a grid with code. The following example updates a grid cell when the user types into a text box:

```
Sub Text1_Change ()
   Grid1.Text = Text1.Text
End Sub
```

Some of the key properties that you will need in order to use this control are as follows.

Property	Default	Comments
ColAlignment	0 - Left Justify	Sets or returns the alignment of data in a column
Cols, Rows		Sets or returns the total number of rows or columns in the grid
ColWidth		Sets or returns the width (in twips) of a specified column
FixedCols, FixedRows		Sets or returns the total number of fixed columns or rows for a grid
GridLines	-1 - **True**	Determines whether lines between cells are displayed
Picture		Sets or returns a graphic for the current cell

3-D Panel

This control has a number of possible uses. It has, however, two primary uses—the first is to provide greater three-dimensional (3-D) quality to another control or group of controls. For example, you can place other controls on the panel; it can be used in place of a frame. You might also use this control as a background for the entire form. The three-dimensional panel, because it has FloodPercent and FloodType properties, can be used as a status or progress indicator.

Some of the key properties that you will need in order to use this control are.

Property	Default	Comments
Alignment	0 - Left Justified Top	Caption alignment for this control offers nine possible choices.
BevelInner	0 - None	Sets the style of the inner bevel of the panel: none, inset, or raised.
BevelOuter	2 - Raised	Sets the style of the outer bevel of the panel
BevelWidth	1	Sets or returns the width of both the outer and inner bevels of the panel.
FloodColor		Sets or returns the color used to paint the area inside the panel's inner bevel when used as a status indicator.
FloodPercent		Sets or returns the percentage of the painted area inside the inner bevel when used as a status indicator.
FloodType	0 - None	Determines whether and how the panel is used as a status indicator.

Walk Through — Cr ating a Perc nt M t r

∑ To set the Percent Meter at work

1. From the Walk Through program group, choose Percent Meter.

   The purpose of this walk through is to demonstrate the status reporting capability of the three-dimensional panel control.

2. On the Percent Meter application, choose Start.

   In a few seconds the panel will progressively change color, indicating along the way the percentage completion of the process.

3. Open the Control menu on the Percent Meter application, and choose Close.

∑ To create the Percent Meter application

1. Start Visual Basic.
2. From the File menu, choose New Project.
3. Set the properties for the form as follows.

Property	Setting
Caption	Percent Meter
Height	4425 (approximately)
MaxButton	**False**
MinButton	**False**
Width	2535 (approximately)

4. From the toolbar, double-click the 3-D Panel tool and set its properties as follows.

Property	Setting
BevelInner	1 - Inset
BevelOuter	2 - Raised
BevelWidth	3
BorderWidth	2
Caption	
FloodColor	Yellow
FloodType	4 - Bottom to Top
Height	2655 (approximately)
Left	240 (approximately)
Outline	**True**
Top	240 (approximately)
Width	1935

5. From the toolbar, double-click the Command button and set its properties as follows.

Property	Setting
Caption	Start
FontSize	13.5
Height	495 (approximately)
Left	480 (approximately)
Top	3240 (approximately)

6. Double-click the Command button in design mode in order to open the Code window.

   Visual Basic displays a Code window with the Command1_Click event procedure template in it.

7. Place the following lines of code between the two parts of the template:

```
Panel3d1.FloodPercent = 0
For i% = 1 To 100
    Panel3d1.FloodPercent = i% * 1
Next i%
```

8. From the Run menu, choose Start.

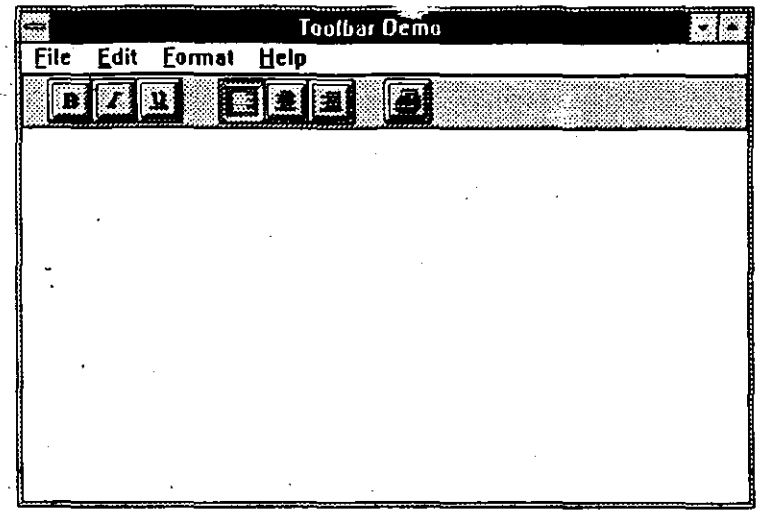
   Test your application.

9. From the Run menu, choose End.

10. Save your form and .MAK file with appropriate names in \WALKTHRU\SAMPLES.

11. Close Visual Basic.

Group Push Buttons



Group push buttons work like combination command buttons and option buttons. They are like command buttons because when the user clicks the button, something happens—a file is opened or saved, or text is right-justified. Group push buttons, however, are also like option buttons because, with the properties set properly, selecting one option can automatically remove the selection from another option.

Some of the key properties that you will need in order to use this control are as follows.

Property	Default	Comments
GroupAllowAllUp	-1 - True	Determines whether all buttons in a logical group can be in the up position.
GroupNumber	1	Sets or returns the group number for a given group push button. This property is used to create logical groups of buttons.
Outline		Sets or returns whether the button has a border around it.
PictureDisabled	None	Specifies the bitmap to display on the 3D group push button when it is disabled.
PictureDn	None	Specifies the bitmap to use when the button is depressed.
PictureDnChange	0	Determines how the PictureUp bitmap is to be used to create the PictureDn bitmap; 2 merely inverts the PictureUp bitmap.
PictureUp	None	Specifies the bitmap to use when the button is up.

Walk Through — Creating a Toolbar

Σ To see the interface you will be creating

1. From the Walk Throughs program group, choose Toolbar.

   This little application doesn't do very much. What it does show is that the user can select any or all of the first three options, whereas they can only select one of the second three. More to the point, it is an example of how you would go about drawing a toolbar like the one found in Microsoft Word for Windows or Microsoft Excel.

2. From the Control menu on the Toolbar form, choose Close.

Σ To create the toolbar

1. Open Visual Basic.
2. From the File menu, choose Open Project.

   Locate TOOLBAR.MAK. It should be located in \WALKTHRU\SAMPLES. This form already contains the basic form—TOOLBAR.FRM with all of its properties already set. In order to add the toolbar, you need to complete the following steps:

3. Place seven group push buttons on the toolbar panel at the top of the form.
4. Use the mouse pointer and the control key to select all seven of the group push buttons, and then set the following properties for all of them.

Property	Setting
Height	420
Top	30
Width	450

5. Place three of the push buttons side by side on the left end of the panel.

   These will become the text formatting buttons for Bold, Italic, and Underline.

6. Select the first button in this group and set the properties as follows.

Property	Setting
GroupNumber	0
PictureUp	BLD-UP.BMP located in \VB\BITMAPS\TOOLBAR3
PictureDn	BLD-DWN.BMP

These changes create the first button on the toolbar and make use of two separate bitmaps to indicate the status of the button to the user.

7. Select the second button in this group and set the properties as follows.

Property	Setting
GroupNumber	0
PictureUp	ITL-UP.BMP
PictureDn	ITL-DWN.BMP

8. Select the last button in this group and set the properties as follows.

Property	Setting
GroupNumber	0
PictureUp	ULIN-UP.BMP
PictureDn	ULIN-DWN.BMP

Notice that all three of these buttons have the same number—0. A group number of 0 means that they are *not* part of a logical group, and therefore changing the state of one will not change the state of others. After all, your user may want to have bold and italic and underlining. Clusters of controls with any other group number will act just like option buttons—only one will be in force at any one time.

For purposes of contrast, we will take a slightly different approach on the second group of push buttons. Here you are giving the user the option of formatting text as either left-, centered-, or right-justified. In this case, you will want these buttons to work as a group because any one bit of text can have only one of these three states.

9. Create the second group of buttons side by side toward the middle of the panel.

In all likelihood, you will not be able to keep the buttons in perfect top-to-bottom alignment. Don't let that bother you. You can select the entire group at the end of this walk through and set the Top property for all of them in one step, as you did above.

10. Select the first button in this group and set the properties as follows.

Property	Setting
GroupNumber	2
PictureUp	LFT-UP.BMP
PictureDnChange	2 - Invert PictureUp Bitmap

Notice what we have done here? We have created the PictureDown bitmap by inverting the image using the property PictureUpChange.

11. Select the second button in this group and set the properties as follows.

Property	Setting
GroupNumber	2
PictureUp	CNT-UP.BMP
PictureDnChange	2 - Invert PictureUp Bitmap

12. Select the third button in this group and set the properties as follows.

Property	Setting
GroupNumber	2
PictureUp	RT-UP.BMP
PictureDnChange	2 - Invert PictureUp Bitmap

What this example emphasizes is that the property GroupNumber is your way of telling Visual Basic how you want individual sets of buttons to be grouped.

13. As a little extra challenge, you set the properties for the last control. It's supposed to represent a printer, so you will need to use the PRT-UP.BMP.
14. Save your work in \WALKTHRU\SAMPLES.

∑ To Code The Toolbar

If you want to implement the toolbar so it functions properly, you can add the following code.

1. Double-click the bold button to open the code window for GroupPush3D1_Click event and add the following code:

```
lblMessage.FontBold = Value
```

2. Double-click the italic button to open the code window for GroupPush3D2_Click event and add the following code:

```
lblMessage.FontItalic = Value
```

3. Double-click the underline button to open the code window for GroupPush3D3_Click event and add the following code:

```
lblMessage.FontUnderline = Value
```

4. Double-click the left-justify button to open the code window for GroupPush3D4_Click event and add the following code:

```
lblMessage.Alignment = 0
```

5. Double-click the center button to open the code window for GroupPush3D5_Click event and add the following code:

```
lblMessage.Alignment = 2
```

6. Double-click the right-justify button to open the code window for GroupPush3D6_Click event and add the following code:

```
lblMessage.Alignment = 1
```

7. Save your work in \WALKTHRU\SAMPLES.

Summary

- **File Browsing**
  - **Grid Control**
  - **3-D Panel Controls and Group Push Button**

Objectives

In this module you learned to:

- Design and develop a form that lets users browse drives, directories, and files.
- Use the Change and PathChange events.
- Use the grid control.
- Use 3-D panel controls and the group push button.

Module 8: Using Visual Basic Data Types

Σ Overview

- Key Terms
- Using Variables and Constants
- Scope
- Additional Visual Basic Data Types

Overview

The overall purpose of this module is to introduce already knowledgeable programmers to the data types of Visual Basic and their scoping rules. This module is designed as the first of a series of modules on the implementation of Basic within Visual Basic.

Prerequisites

Prior to starting this module, you should already be familiar with:

- Data types and scoping rules within some other procedural language
- The general Visual Basic programming system

Overall Objective

At the end of this module, you will be able to correctly and efficiently use almost all of the Visual Basic data types.

Learning Objectives

At the end of this module, you will be able to:

- List and describe the seven variable data types in Visual Basic.
- Correctly identify the six type declaration character symbols.
- Use the contents of CONSTANT.TXT.
- Distinguish among the various levels of scope in V al Basic.
- Explain the rules for using each level of scope.
- Declare and use a user-defined data type.
- Declare and use simple arrays.

Key Terms

- Variables
- Constants
- Procedures
- Statements
- Scope
- Module

Variables Changeable values that the program manipulates

Constants Unchanging values that the program manipulates

Procedures Activities that the program performs

Statements Subactivities within procedures

Scope (Accessibility) Which part of a program can access specific data or procedures

Module File containing code and data not attached to a form

∑ Using Variables and Constants

- Data Types of Variables
- Data Types of Constants

Data Types of Variables

- Integer %
- Long &
- Single !
- Double #
- Currency @
- String $
- Variant (none)

There are seven fundamental data types of variables that you can use in Visual Basic.

Type name	Description	Type-declaration character	Range
Integer	Two-byte integer	%	–32,768 to 32,767
Long	Four-byte integer	&	–2,147,483,648 to 2,147,483,647
Single	Four-byte floating-point number	!	–3.40E+38 to 3.40E+38
Double	Eight-byte floating-point number	#	–1.79D+308 to 1.79D+308
Currency	Eight-byte number with a fixed decimal point	@	–9.22E+14 to 9.22E+14
String	String of characters	$	0 to 65,500 characters (approximately)
Variant (Default)	Date/time, floating-point number, string	(none)	Date values: January 1, 0000, to December 31, 9999; numeric values same as **double**; string values same as **string**

Declaring Variables

In polite society you always formally introduce strangers to each other the first time they meet. Visual Basic is no exception. You should introduce your variable to Visual Basic by declaring the name of your variable before you use it. You do this in one of two ways: by using the **Dim** statement, or by using one of the two keywords—**Global** and **Static**. As you will see in a couple of pages, not declaring variables can be risky business in some situations because the default data type in Visual Basic is **Variant**.

You have two choices on how to explicitly declare variable data types.

Using As	Using the type-declaration character
`Dim I As Integer`	`Dim I%`
`Dim Amt As Double`	`Dim Amt#`
`Dim YourName As String`	`Dim YourName$`
`Dim BillsPaid As Currency`	`Dim BillsPaid@`

In order to make compound variable declarations, you merely state:

```
Dim X as Integer, Y as Long
```

Caution If you declare three separate variables with the following statement, you will get K$ as a string and I and J as variants, not as strings.

```
Dim I, J, K$
```

Explicit Declarations

To require explicit declaration of variables in all of your code, place the following statement in the General Declarations section of any form or module.

```
Option Explicit
```

If you want to require that all variables be explicitly declared in all of your projects, from the Options menu choose Environment and set the Require Variable Declaration option to YES.

Variable-Length Strings

Sometimes you may need a variable that can hold strings of different lengths at different times. This is called a variable-length string variable. You declare it like this:

```
Dim Message As String
```

Fixed-Length Strings

At other times you know that the strings that will be assigned to a certain variable will never have more than a certain number of characters. In this case, you can declare a *fixed-length* variable like this:

```
Dim FixedLengthString As String * 50
```

In this example, you are declaring a string variable with the name FixedLengthString, and you are telling the application that the variable may contain up to 50 characters.

You can take this one step further. For example, set the length using a **Global** constant with the name FixedLength, set the value of the constant to 50, and then point to that constant name using the following syntax:

```
Global Const FIXEDLENGTH = 50
Global FixedString As String * FIXEDLENGTH
```

Rules When Working with Variables

1. Variable names can be up to 40 characters long.
2. Names can include only letters, numbers, and underscores.
3. The first character in the name must be a letter.
4. You cannot use Visual Basic reserved words. See the list of programming topics in Visual Basic Help for a partial list of reserved words.

Variant Data Type

As the chart above shows, the **Variant** data type is the default. The **Variant** data type can store numeric, string, or date/time information. You don't need to convert between these kinds of data when assigning them to a variant variable; Visual Basic automatically performs any necessary type conversions for you.

Rules for Using Variant Variables

1. If you perform arithmetic operations or functions on a variant, it must contain a number. If you want to test a variant to see if it contains numeric data, use the **IsNumeric** function.
2. Normally, when you concatenate two strings you would use the plus (+) sign. To avoid ambiguity with variant data, use the ampersand (&) to indicate concatenation.

Note When concatenating, be careful to leave a space between a variable name and the ampersand. If you don't, Visual Basic assumes that you are typecasting the variable into a **Long** data type.

---

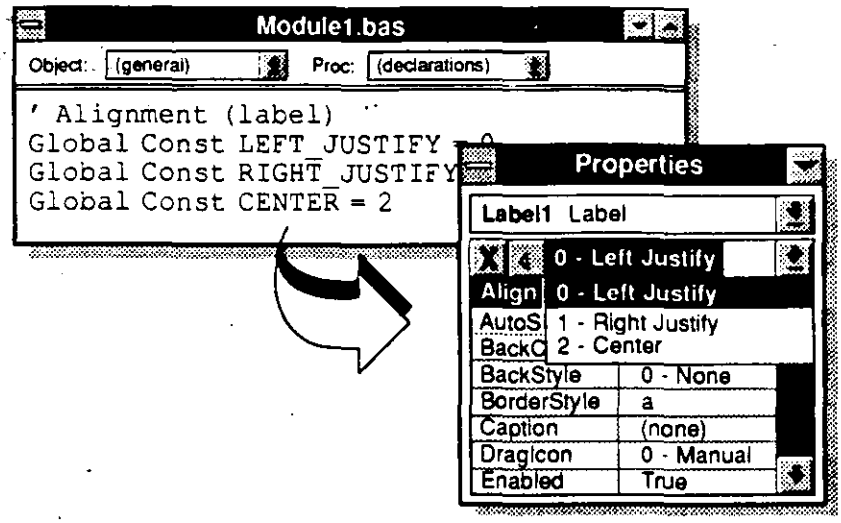
3. When passing a **Variant** variable as an argument, check the procedure parameters. If the corresponding argument is an explicit data type, you must pass the **Variant** variable with parentheses around it in order to pass it by value.

```
Dim V As Variant
V= "Testing"
Debug.Print PrintString((V))      'Extra parentheses to pass by value

Sub PrintString(S As String)
    'Do something
End Sub
```

4. To determine the internal representation for a variant variable, use the **VarType** function. See documentation for specific return values.
5. A variant variable has the **Empty** value before it is assigned a value. The **Empty** value is a special value, different from a zero, a zero-length string, or a NULL value. To test your **Variant** variable for an **Empty** value, use the **IsEmpty** function. To reassign a **Variant** variable back to the **Empty** value, you must assign another empty **Variant** variable to it.
6. To test for a **Variant** variable containing NULL, use the **IsNULL** function.

Note For a listing of the string conversion functions for the **Variant** data type, see the appropriate table at the end of the Visual Basic code module.

Constants



What Are Constants?

Constants are just that, entities within your program whose value you need, but which will not be changed during the running of the program. For example, if your program needed to work with circles, you would probably need a constant like PI = 3.1459.

Examples of typical constants can be found in CONSTANT.TXT. If you look at this file, you will find several constant values declared that are familiar to you from the overview of properties for controls.

Example

```
1   'Border Style
2   Const NONE = 0
3   Const FIXED_SINGLE = 1
4   Const SIZABLE = 2
5   Const FIXED_DOUBLE = 3
```

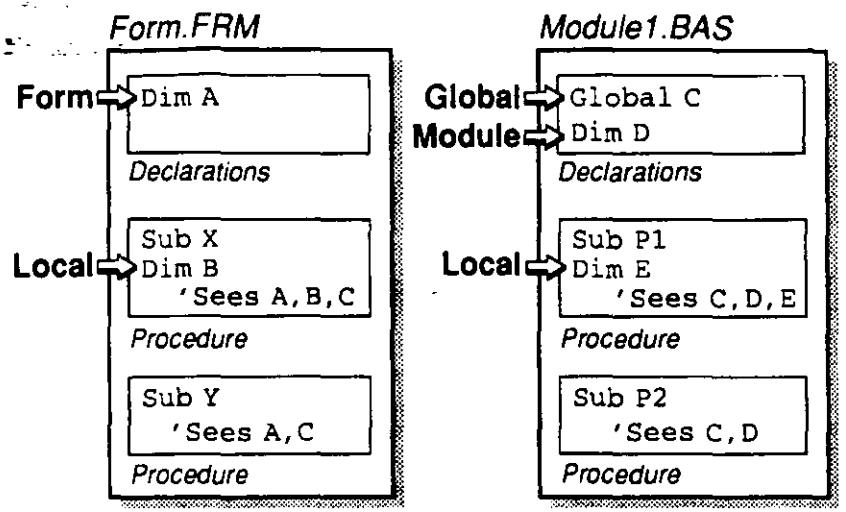
How Are They Declared?

Visual Basic contains a CONSTANT.TXT file, which lists all the common constants. You can copy and then modify this file to meet the application's specific needs. You can then load CONSTANT.TXT into the General Declarations section of a module (.BAS) to be used by the application. Following is a partial listing of the general categories of constants defined in CONSTANT.TXT.

∑ Scope

- Scope of Data
  - Declaring and Using Local Variables
  - Declaring and Using Form-Level and Module-Level Variables
  - Declaring and Using Global Variables

Scope of Data



What Is Scope of Data?

The scope of a data variable or constant is its level of visibility within an application. Scope of data falls into three levels of visibility: local variables, form- and module-level variables, and global variables.

Scope of Data — An Analogy

Scope of data basically deals with who can use what. The easiest way to think of this is by analogy.

Say you had a refrigerator/freezer full of all kinds of goodies—candy, soft drinks, ice cream. Almost everyone likes these goodies and would like to get to them. Your job as the manager of these resources is to make sure that the right people get to them and the wrong people don't.

If you put these goodies out where everyone can get to them, they will be used, right? If that is what you want, then fine, make them *globally* available. In Visual Basic terminology, declare and define variables that all functions will need access to using the **Global** keyword in the General Declarations section of a module (.BAS), not a form.

For those goodies that you only want some people to get to, for example, just the members of your work group, then put the refrigerator/freezer where only they can get to it—for example, in your work group area. In Visual Basic, declare and define these variables using the keyword **Dim** in the General Declarations section of either the form or the module. You'll see an example of how this is done in just a few minutes.

Finally, for those goodies to which only you should have access, put the refrigerator in your office. That is, declare and define those variables locally—inside a procedure. Again, you'll see examples of this is just a minute.

Variable Declaration Keywords and Their Scope

The table below details how the different variable declaration keywords are used.

Scope	Declaration
Local	**Dim, Static,** or **ReDim** (within a procedure)
Module	**Dim** (in General Declarations section of a form or a code module).
Global	**Global** (in General Declarations section of a module)

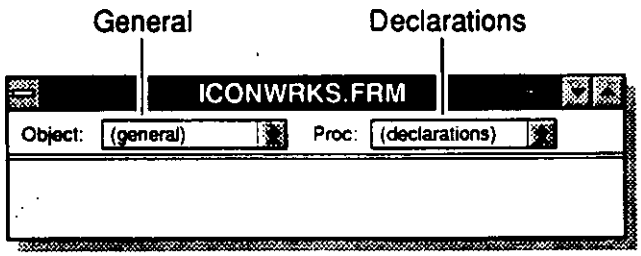
Σ A simple example of data declarations can be found this way

1. Start Visual Basic.
2. From the File menu, choose Open Project.
3. Go to the C:\VB\SAMPLES\CALC subdirectory.
4. Double-click CALC.MAK.
5. Click CALC.FRM in the Project window.
6. Click View Code in the Project window.

Visual Basic displays the form-level variable declarations for the calculator application. Notice the declaration of seven variables and the **Option Explicit** statement.

Note When you are writing code and you want to create module- or form-level variables, make sure that you are really in the general declarations section of the code before you make changes or additions.

Always be sure that the top of your form window has these two options selected.



Σ **For a more complex listing of global declarations**

1. From the File menu, choose Open Project.
2. Go to the C:\VB\SAMPLES\ICONWRKS subdirectory.
3. Double-click ICONWRKS.MAK.
4. Click ICONWRKS.GBL in the Project window.
5. Click View Code in the Project window.

Scroll down to the General Declarations section of ICONWRKS.GBL and inspect the various values declared and defined.

Rules for Using Local Variables

1. A local variable is recognized and declared only within the procedure in which it appears using the keyword **Dim**. The variable is re-initialized to zero or a NULL string when the procedure begins, unless the local variable is declared as **Static**. A local variable is a good choice for any kind of temporary calculation.
2. If you use a variable without declaring it, then Visual Basic assumes it is local and assumes **Variant** as the data type. However, this technique may waste storage space and is not as reliable as declaring the variable in the appropriate procedure. It is also not as efficient.

Rules for Using Form- and Module-Level Variables

1. A form-level variable is declared in the General Declarations section of the form, using the keyword **Dim**. A form-level variable is available to any procedure on that form and only that form.
2. A module-level variable is declared in the General Declarations section of the module, using the keyword **Dim**. A module-level variable is available to any procedure in that module and only that module.

Rules for Using Global Variables

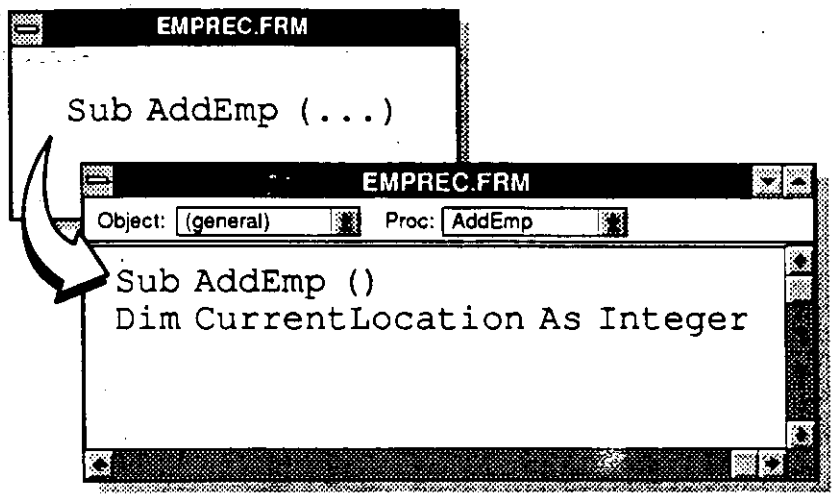
1. A global variable or constant is shared throughout an application. Multiple forms and modules can use a global variable. They are declared in the General Declarations section of a module using the **Global** keyword for variables and **Global Const** for constants.
2. Global variables are persistent; they retain their value throughout the entire application.

Style Guidelines

Good programming practices when using variables and constants include:

- Use form-level and module-level constants and variables as opposed to global-level if possible.
- Declare the variable or constant with the **Dim** statement rather than accepting Basic automatic variables.
- Use **Integer** declarations instead of default **Variant** declarations whenever possible. **Integer** declarations take less space, avoid rounding errors, and consume less CPU time than **Variant** declarations.

Declaring and Using Local Variables



Local variables reset to value zero (0) or an empty string when the local procedure begins.

To declare a local variable, place the **Dim** data declaration statement inside the local procedure.

Example

```
1   Sub Form_Click ()
2       'The local variable below disappears upon procedure exit
3       'and is reinitialized to zero when the procedure is re-entered
4       Dim EmployeeNumber As Integer

5       EmployeeNumber = Val(Text1.Text)
6       Print "The Employee Number is:  " + Str$(EmployeeNumber)
7   End Sub
```

In the example above, EmployeeNumber is declared as an **Integer**. How would the code be different if you handled it as a **Variant**?

Example

```
1   Sub Form_Click()
2       Dim EmployeeNumber

3       EmployeeNumber = Text1.Text
4       Print "The Employee Number is: " & EmployeeNumber
5   End Sub
```

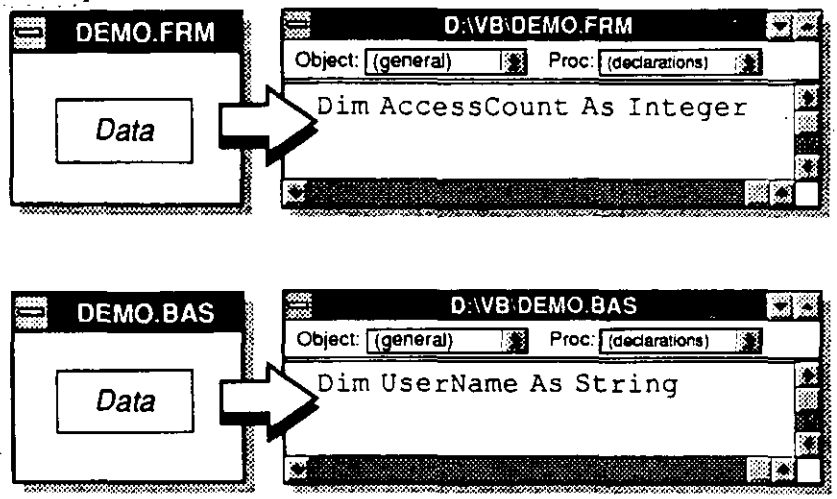
Note Notice the use of the ampersand to concatenate the literal **String** and the **Variant** in the **Print** statement above?

If you want a local variable to be persistent, declare it as **Static**. **Static** local variables will retain their data value when the procedure performs a return.

Example

```
Sub StaticDemo ()
    'The static variable below will retain its value across
    ' calls and is used to keep track of how many times
    ' StaticDemo is called.
    Static RunCount As Integer  ' Create a persistent variable
    RunCount = RunCount + 1     ' Use it to keep run count.
    Msg$="The RunCount for the StaticDemo procedure is: "
    Msg$ = Msg$ + Str$(RunCount)
    Print Msg$                  ' Display message.
End Sub
```

Declaring and Using Form-Level and Module-Level Variables



Use form-level variables or module-level variables when sharing data only within a form or a module. The data remains persistent within the form or module but is not accessible to procedures in other forms or modules.

To declare a form-level or module-level variable, place the **Dim** statement in the General Declarations section of the corresponding form or module.

The declaration below appears in the General Declarations section of the module DEMO.BAS:

```
Dim UserName As String
```

When the procedure below is called, it can access this module-level variable. For example, suppose that a text box is supplied for the user to type his or her name:

```
    .
    .
UserName = txtUserName.Text
```

Then the appropriate value for UserName will be printed when the **Sub** procedure WelcomeDemo is called:

Example

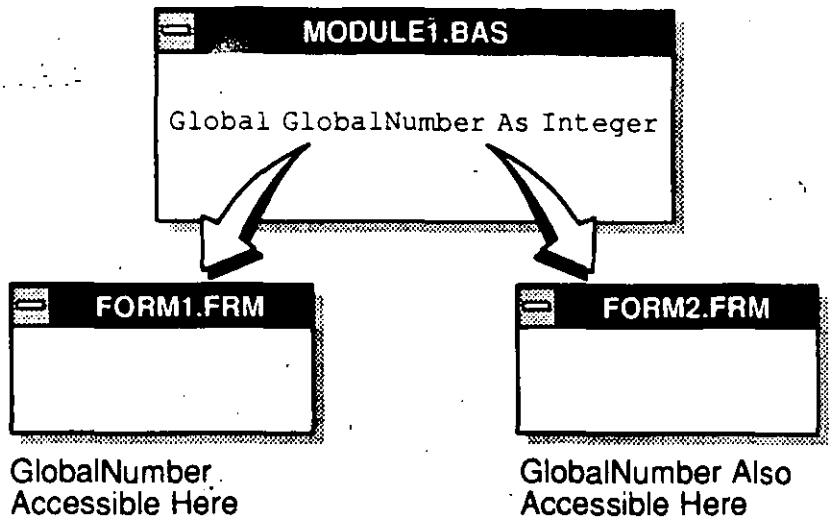
```
1   Sub WelcomeDemo ()
        .
        .
2   ' Variable declared in module's Declarations section can be
3   ' referenced here
4       Print "Welcome " + UserName
5   End Sub
```

The module-level variable is also available to all other procedures defined in DEMO.BAS.

Example

```
1   Sub VerifyUserName ()
2       Msg$ = "UserName = "
3   ' Variable declared in module's Declarations section can
4   ' also be referenced here
5       MsgBox Msg$ + UserName
6   End Sub
```

Declaring and Using Global Variables



Use global variables and constants when sharing data throughout an application. The data remains persistent and accessible from all forms and modules within the application.

To declare a global variable, place the **Global** statement in the General Declarations section of a module—that is, a .BAS file:

```
Global GlobalNumber As Integer
```

The variable GlobalNumber can now be referenced throughout the application.

Example 1

```
1   ' Code for Form1
2   Sub Form_Click ()
3   ' Global variables can be referenced here
4       GlobalNumber = GlobalNumber + 1
5   End Sub
```

Example 2

```
1   ' Code for Form2
2   Sub Form_Click ()
3   ' Global variables can also be referenced here
4       Print = "GlobalNumber " + Str$(GlobalNumber)
5   End Sub
```

Load any constants you want declared for an entire program into the General Declarations section of a module.

Walk Through — Scope

Σ To demonstrate the concepts of scope of data in Visual Basic

1. From the Walk Throughs program group, start Scope.

   When the application starts, you see two forms. Form 1, on the left, contains a Command1 button. Form2, on the right, contains a Command1 button.

   The purpose of this walk through is to demonstrate the concepts of scope of data in Visual Basic.

2. Click Form1 (on the form, not the command button).

   The Form_Click event will display values of data items as their scope permits.

   The global and form-level data values are displayed. Note, however, that the Form2 module-level variable and the Form1.Command1_Click and Form2.Command1_Click local variable values are not displayed. Remember, you just requested a Form_Click event, so the Command1_Click variables are not yet in scope and Form2 variables are not in scope until Form2 has focus.

3. Click Form2 (on the form, not the command button).

   The Form_Click event will display values of data items as their scope permits.

   The global and form-level data values are displayed. Note, however, that the Form1 module-level variable and the Form1.Command1_Click and Form2.Command1_Click local variable values are not displayed either. Remember, you just requested a Form_Click event, so the Command2_Click variable is not yet in scope and Form1 variables are not in scope until Form1 has focus.

4. Click the Command1 button on Form1.

   The Command1_Click event will display values of data items as their scope permits.

   You have just generated a Form1.Command1_click event, and this places the Form1.Command1_Click variable in scope. Form1 still has focus, though, so Form2's variables are not accessible.

5. Click the Command1 button on Form2.

   The Command1_Click event will display values of data items as their scope permits.

   You have just generated a Form2.Command1_Click event, and this places the Form2.Command1_Click variable in scope. Form2 still has focus, though, so Form1's variables are not accessible.

6. Double-click the Control menus on both forms to close them.

Additional Visual Basic Data Types

- User-Defined Data Types
- Arrays

User-Defined Data Types

Earlier in this module standard Visual Basic data types were discussed. Visual Basic also allows you to create your own data types. User-defined data types, called records or structures in some other programming languages, must be declared in the General Declarations section of a module, using the **Type** statement.

Syntax

Type usertype

elementname As typename

[elementname As typename]

.

.

.

End Type

This feature allows you to create new variables that can be customized to fit the needs of your application. For example, you might want to define a data type to hold customer information as follows:

Example

```
1  ' Place this declaration in a module
2  Type CustomerRecord
3      CustNum As Long
4      CustName As String * 38
5      CustAddress1 As String * 38   ' Street Address
6      CustAddress2 As String * 38   ' City, State
7      CustZip As String * 10
8  End Type
```

You can then declare and use variables of this new type in your application wherever you need them. To refer to a particular element, use the syntax *variablename.elementname*.

Example

```
' Sub GetCustomerInfo ()
' Declare a variable of your user-defined type
Dim NewCustomer As CustomerRecord

' Use contents of various text boxes on a form to fill in values to
' the elements (fields) of your new CustomerRecord variable
NewCustomer.CustNum = Val(txtNumber.Text)
NewCustomer.CustAddress1 = txtAddr1.Text
NewCustomer.CustAddress2 = txtAddr2.Text
NewCustomer.CustZip = txtZip.Text

End Sub
```

Arrays

Visual Basic, like many other programming languages, allows you to create arrays. An array is a group of variables of the same data type that share a common name. Each separate element of the array is identified by a unique index number.

Example

```
' Declare an array
Dim ArrayName(UpperBounds) As DataType
Dim TestScores(23) As Single
```

Note By default, the first element in the array is referred to with the index 0 (zero). This creates an array of 24 (not 23) single-precision numbers in which you could store the scores for 24 students.

If you wanted to print this score out to the form, you would use the following:

Example

```
Print "The test score for the first student is: "
Print TestScores(0)
```

There is an alternate syntax for declaring arrays that allows you to specify the lower (beginning) and upper (ending) indexes, or bounds, of an array. You could declare an array to hold 24 test scores like this:

```
Dim TestScores(1 To 24) As Single
```

You would then refer to the first element in the array as TestScores(1).

If you didn't have arrays, you would have to create 24 separate variables, named something like Score1, Score2, Score3, and so on.

In the module on looping structures, you will discover that loops are a very useful tool for accessing elements in an array.

Note To change the default lower bound to 1, place an **Option Base** statement in the General Declarations section of the module, like this:

```
Option Base 1
```

Variations on Array Declaration Syntax

Depending upon where the array is declared and the special needs of your program, the syntax to declare an array may vary a bit. Here is a table that summarizes the rules.

Scope of variable declaration	Keyword to use
Application-wide	**Global** (Used only in the General Declarations section of a .BAS module.)
Form or module level	**Dim** (Used in the General Declarations section of a form or module.)
Within a procedure	**Static** (If the entire procedure has been declared **Static**, then you may use the word **Dim** to declare the array.)

Multidimensional Arrays

The array described in the last section, TestScores, is an example of a one-dimensional array. You could think of it as representing a single column (or row) of numbers. You can also declare an array of several dimensions. The maximum number of array dimensions allows in a **Dim** statement is 60. For example, to represent a table of numbers you can declare a two-dimensional array as follows:

```
Sub GetNumbers ()
   Static Table(4, 23) As Single

   -Or-

   Static Table(1 To 5, 1 To 24) As Single

End Sub
```

This would create an array of five rows of 24 test scores each and provide room to hold five scores for each of 24 students.

Dynamic Arrays

There are times when you want to use an array, but the size needed will change at run time. Visual Basic allows you to create dynamic, or variable-length arrays by doing the following:

Declare the array without declaring its size with either the **Dim** statement (for a form- or module-level array) or the **Global** statement (for a global array). To do this, just place an empty set of parentheses to the right of the array name:

```
' Place in General Declarations section of a form or module
Dim DynArray() As String * 25

-Or-

Global DynArray() As String * 25       ' Place in MODULE1.BAS
```

Here is another example:

```
Global DynIntArray() As Integer
```

Then inside a procedure, when you know the size you need for this array in some particular circumstance, redimension the array to the size you need:

Example

```
Sub UseArray ()
    .
    .
' Notice you can use a variable from your program to set the size
ReDim DynArray( List1.ListCount)
    .
    .
End Sub
```

Important Each time you use the **ReDim** statement, all the values currently stored in the array are lost, and each element is reset to zero or a NULL string depending on the type of the elements in the array. If you want to preserve the values during re-dimensioning of the array, use the **Preserve** keyword.

```
ReDim Preserve MyArray(UBound (MyArray) + 10)
```

In this case, you are re-dimensioning an array called MyArray to be 10 elements larger while maintaining existing data.

Control Arrays

Visual Basic has a special type of array called a control array, which has special features and does not follow all the rules of standard arrays.

You will learn to use control arrays in the course that follows this one in the Microsoft University Visual Basic curriculum.

Summary

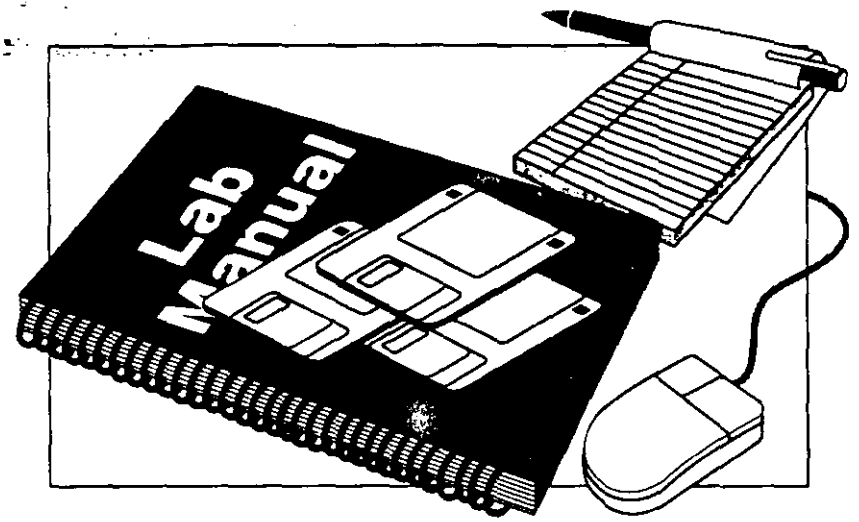
- Key Terms
- Using Variables and Constants
- Scope
- Additional Visual Basic Data Types

Objectives

In this module you learned to:

- List and describe the seven variable data types in Visual Basic.
- Correctly identify the six type declaration character symbols.
- Use the contents of CONSTANT.TXT.
- Distinguish among the various levels of scope in Visual Basic.
- Explain the rules for using each level of scope.
- Declare and use a user-defined data type.
- Declare and use simple arrays.

Lab Time



Go to the Using Constants and Variables portion of your lab manual.

Module 9: Writing Visual Basic Code

Overview

- Visual Basic Procedures
- Scope of General Procedures
- Writing Code in Visual Basic
- String and Numeric Conversion Functions

Overview

This is the "Everything You Ever Wanted to Know About Writing Code...But Were Afraid to Ask" module. It gives you the background and procedures needed to begin writing the Visual Basic code that implements the interfaces.

Prerequisites

To successfully complete this module and its associated lab, you should have a detailed understanding of forms, controls, and properties as they are implemented in Visual Basic.

Knowledge of any block structured programming language is required.

Overall Objectives

There are two overall goals for this module:

1. To quickly review most of the fundamental procedures of Basic as it is used in the Visual Basic product
2. To show students the first steps required for completing the code for an application

Learning Objectives

At the end of this module, you will be able to:

- Define important characteristics of **Functions** and **Sub** procedures and write code that correctly uses both.
- Define and correctly code for appropriate scope of data.
- Write code that uses any of the common Basic data types.

Sample Code

The best way for you to learn how to write Visual Basic code is to read it. Sample code for a four-function calculator application can be found in the \VB\SAMPLES\CALC subdirectory.

Σ Visual Basic Procedures

- **Sub Procedures**
- **Functions**
- **Arguments and Parameters**
- **Procedures**

  Event Procedures

  General Procedures

  Methods

What Are Procedures?

A procedure is a block of Visual Basic statements that are called as a logical unit.

Two Types of Procedures — A First Pass

Sub Procedures

After the **Sub** procedure completes its work (executes its code), it returns to the procedure that called it.

Syntax

```
Sub SubName ()
    statementblock
End Sub
```

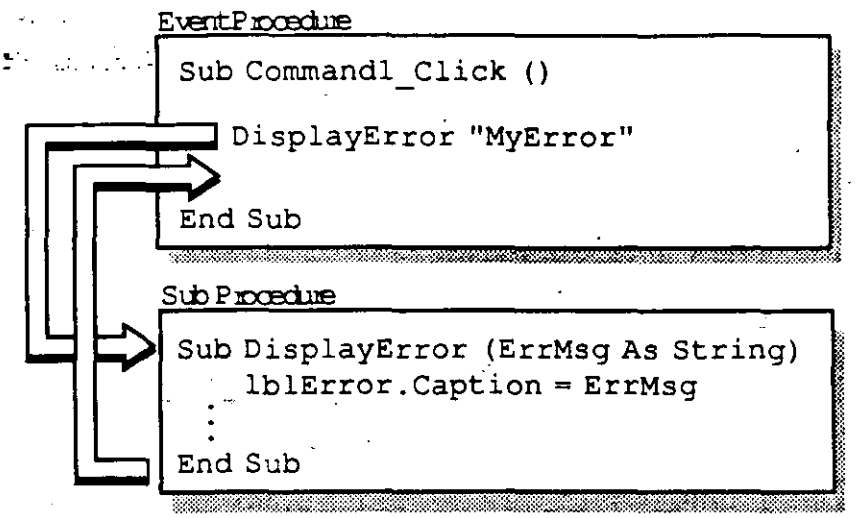
Functions

A **Function** is similar to a **Sub** procedure. In addition, it has a data type just as a variable does. After it completes its work, it returns a value of that type to the procedure that called it.

Syntax

```
Function FunctionName() As SomeDataType
    statementblock
    FunctionName = SomeValue
End Function
```

Calling a Sub Procedure with Arguments



Arguments

Any procedure can be defined to receive data when it is called. A piece of data sent to a procedure is called an argument; and an argument is matched to its equivalent parameter entry in the procedure list. Each parameter is declared to be of a specific data type.

The following is a **Sub** procedure that receives two arguments:

Syntax

Sub *SomeSub* (*Param1* **As Integer**, *Param2* **As Single**)

statementblock

End Sub

When you call this procedure you must make sure to pass it the same number and type of arguments in the same order they appear in the **Sub** definition.

Syntax

SomeSub Argument1, Argument2

–Or–

Call *SomeSub* (*Argument1, Argument2*)

The following calls a **Function** procedure with arguments.

Syntax

Dim *Result* **As Integer**

Result = *SomeFunction* (*Argument1*, *Argument2*)

Function *SomeFunction* (*Param1* **As Integer**, *Param2* **As Single**)
^ **As Integer**

StatementBlock

SomeFunction = *ReturnValue*

End Function

Note The argument names used when you call the procedure do not have to match the argument names used in the definition of the procedure as they do in the above example. The number, data type, and order of the arguments must match in the procedure definition and call.

An alternative notation for the function return value data type is as follows.

Syntax

Function *SomeFunction%* (*Param1* **As Integer**, *Param2* **As Single**)

StatementBlock

SomeFunction = *ReturnValue*

End Function

Note **Function** procedures with an explicit return value data type are more efficient than those with a **Variant** return value data type.

Remember what we said about **Variant** data types earlier? If your argument is a **Variant** and its corresponding parameter is not, the **Variant** argument must be passed by value. This is accomplished by putting a set of extra parentheses around the **Variant** argument.

You can accomplish the same objective by using the **ByVal** keyword when you declare your parameter in the procedure.

Example

```
1   Function Reverse (S As String, ByVal N As Integer) As Variant
2   '...SomeStatements
3   End Function

4   Dim U As Variant, V As Variant, W As Variant
5   V = "Testing"
6   W = 10
7   U = Reverse ((V), W)
```

Passing Arguments by Value

Visual Basic passes arguments by reference as a default. What this means is that the procedure can modify the values of the arguments in the procedure list because it knows the address for the data. This in effect allows you to pass back more than one value merely by changing the arguments within the procedure.

In contrast, you can pass an argument to a procedure by value, which means that the procedure receives only a copy of the data and cannot modify the value of the actual argument. Any changes made to the argument within the procedure are local and have no effect on the actual data.

To pass an argument by value, use the keyword **ByVal** in the parameter list or put a set of parentheses around the argument in the calling statement.

Types of Procedures

- Event Procedures
- General Procedures

Types of Procedures — A Second Pass

Visual Basic applications have two categories of procedures: event and general.

Event Procedure

An event procedure is a procedure invoked by a user- or system-triggered event.

Event procedures are always attached to a given form or control. The first part of an event procedure's name indicates which object it is attached to.

Syntax

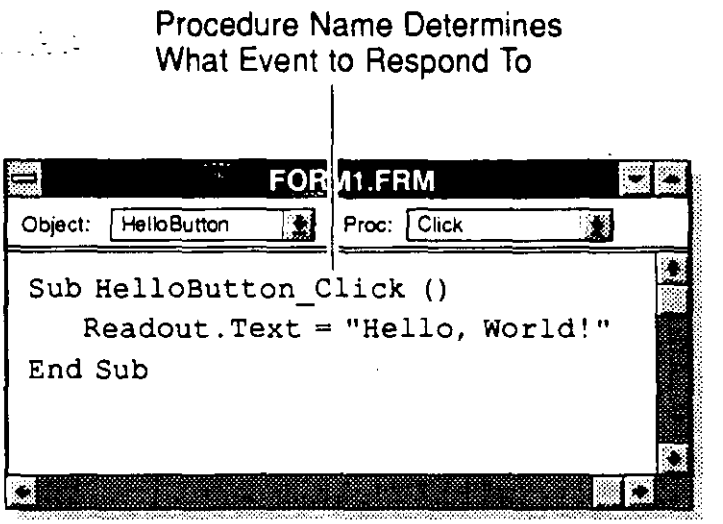
Sub *objectname_eventname* ()

statementblock

End Sub

Examples of event procedures are: Command1_Click and Form_Click. If the user clicks on the command button named Command1, the event procedure Command1_Click will be called; but if the user clicks on the *form*, the event procedure Form_Click will be called.

Creating the Event Procedure



Templates (**Sub** and **End Sub** statements) are supplied for all the events Visual Basic automatically recognizes.

All you need to do is fill in the code:

1. Open the Code window.
2. In the Object list box, select the appropriate object.
3. In the Procedure list box, select the appropriate event.
4. Type code into the template provided by Visual Basic.

Calling the Event Procedure

You don't have to do anything to call the event procedure. Visual Basic automatically recognizes all the events for all Visual Basic objects (forms and controls). As soon as a user or the system triggers an event for an object, the code in the appropriate event procedure will be run by Visual Basic.

Scope of Event Procedures

Event procedures are only available on the form where they were defined.

Note Event procedures may only be **Sub** procedures, not **Function** procedures.

Example

The following is a Click event procedure for a command button.

```
Sub Command1_Click ()
    ' This value can be found in CONSTANT.TXT
    Const RED = &HFF&
    ' Set the background color of the form to red
    Form1.BackColor = RED
End Sub
```

Event Procedures in Visual Basic

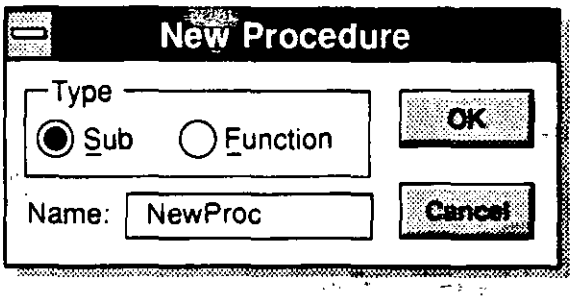
A large number of event procedures are available for your use in Visual Basic. The ones set in italic type below are covered in this course, and the remaining event procedures are covered in the *Programming in Microsoft Visual Basic 3.0* course.

Action	Event
Change to control	***Change***
	DropDown (combo box only)
	PathChange, ***PatternChange*** (file list box only)
Drag and drop	**DragDrop**, **DragOver**
Dynamic data exchange (DDE)	**LinkClose**, **LinkError**, **LinkExecute**, **LinkOpen**
Keystroke	**KeyDown**, **KeyUp**, **KeyPress**
Mouse	***Click***, ***DoubleClick***, **MouseDown**, **MouseUp**, **MouseMove**
Shift in focus	**GotFocus**, **LostFocus**
Timer interval	***Timer***
Forms and picture	***Paint***, ***Resize***, ***Load***, ***Unload***

References

Use the online Help or the *Microsoft Visual Basic Language Reference* to find descriptions of the event procedures in Visual Basic.

General Procedure



A general procedure is a procedure executed only when explicitly called by another procedure.

Creating the General Procedure

1. Open the Code window.
2. From the View menu, choose New Procedure.
3. Type the name for the procedure, and choose either Sub or Function.
4. Add the code to the procedure template provided by Visual Basic. If you are creating a new function, remember to define the return type of the function and to include a statement in the code assigning the value you want returned to the function name.

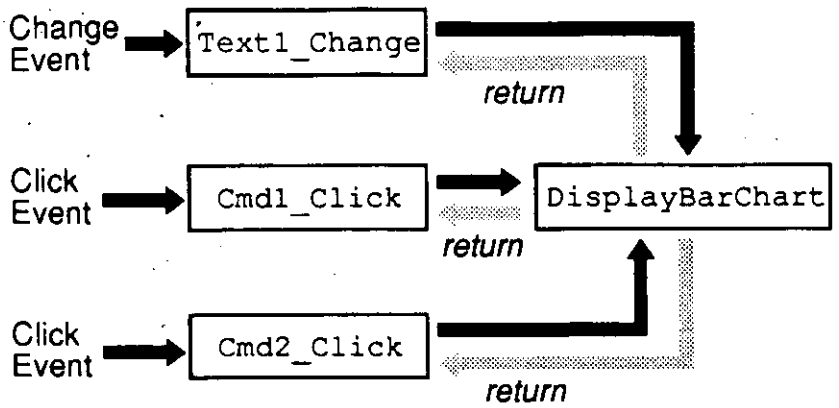
Example

```
1   Function StockValue (NumShares As Integer, SharePrice As Single)
2   ^ As Single
3       StockValue = NumShares * SharePrice
4   End Function

5   Sub TotalStock ()
6       Print "Total value of all stock is: "
7       Print  StockValue(10, 123.25) + StockValue(200, 19.75)
8   End Sub
```

Calling the General Procedure

You must explicitly call a general procedure, or the code will never be run. Typically, you place a call to a general **Sub** or **Function** inside an event procedure or inside another general procedure.

Scope of General Procedures



Form-Level Scope of Code (.FRM)

The scope of a general procedure depends on where it is defined. If you follow the steps listed above while you are in the Code window for one of your forms, then that procedure is available from anywhere on that form only.

Global Scope of Code (.BAS)

If you have a multiple-form application and need to be able to call certain procedures from anywhere in your application, you will need to create a separate .BAS module to hold those procedures.

Note, for example, if DisplayBarChart is defined as a general procedure for Form1, then Text1, Cmd1, and Cmd2 must all be controls on Form1. If, however, DisplayBarChart is defined in a separate .BAS module, Text1, Cmd1, and Cmd2 could each be on a different form and still be able to call DisplayBarChart.

Private Scope of Code (.BAS)

In some cases, you will want to limit the accessibility of a procedure contained within a module. In order to do this, use the **Private** keyword in front of **Sub** or **Function** definitions.

Declaring a Private Sub Procedure

Syntax

Private Sub *SomeSub* ()

SomeStatements

End Sub

Declaring a Private Function

Syntax

Private Function *SomeFunction* () **As Integer**

SomeStatements

End Function

Method

A method is a special type of procedure provided for you by Visual Basic for specific objects but not associated with a specific event.

Special Characteristics of a Method

1. You cannot create a method; you can only call it.
2. You cannot view or change the code for a method.
3. The names of all Visual Basic methods are keywords. You cannot create a general procedure of your own with the same name as a Visual Basic method.

The following calls a method from an event procedure attached to a form:

```
Sub Form_Click()                ' Form Click event for Form1
    Form2.Show                  ' Display Form2
End Sub
```

Methods in Visual Basic

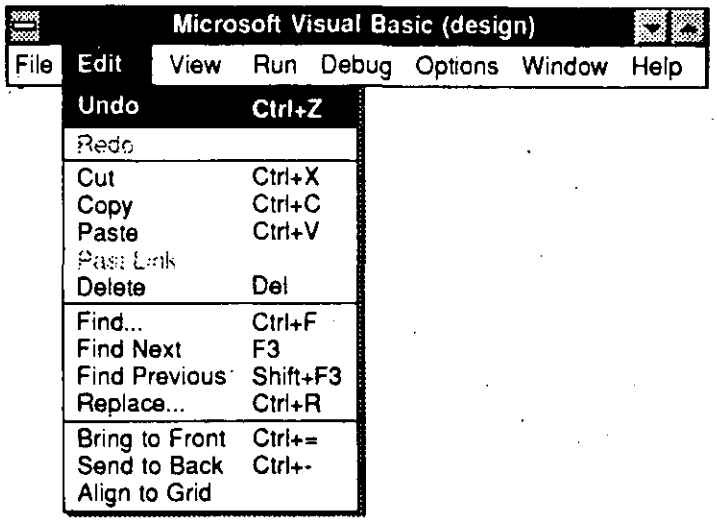
A large number of methods are available for your use in Visual Basic. The ones set in italic type below are covered in this course.

Drawing and graphics	**Circle, Cls, Line, Point, Pset**
Printing	**EndDoc, NewPage, *Print*, *PrintForm*, TextHeight, TextWidth**
DDE	**LinkExecute, LinkPoke, LinkRequest, LinkSend**
List box management	***AddItem*, *RemoveItem*, *Clear***
Clipboard	**Clear, GetData, GetFormat, GetText, SetData, SetText**
Moving controls	**Drag, Move**
Form management	***Hide*, *Show*, Refresh, Scale, SetFocus**

References

Use the online Help on the *Microsoft Visual Basic Language Reference* to find descriptions of the methods available for Visual Basic objects.

Writing Code in Visual Basic



There are several facilities built into Visual Basic that make writing code easy. All of the search and replace procedures discussed below let you search in the current procedure only, in the current module only, or in all modules.

- Cutting, copying and/or pasting code

  Edit menu, Cut or Copy, and Paste
- Finding strings (variable names, for example) in code

  Edit menu, Find
- Finding the next instance of a string in code

  Edit menu, Find Next
- Finding the previous instance of a string in code

  Edit menu, Find Previous
- Finding and replacing a string in code

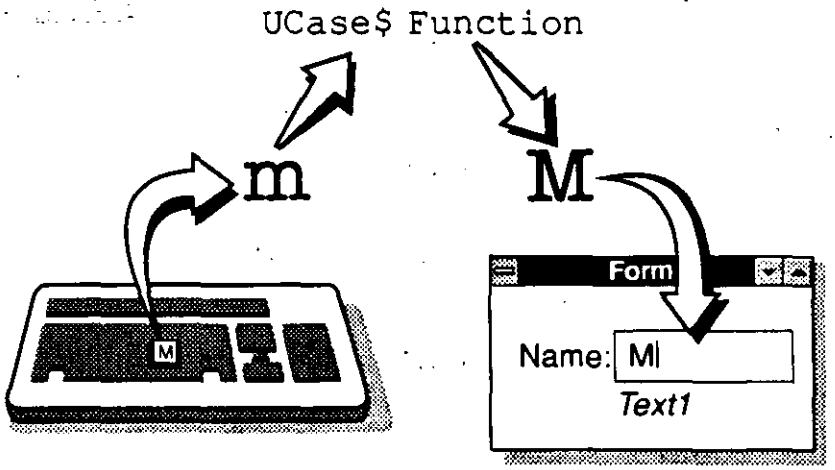
  Edit menu, Replace
- Loading text from the hard disk

  File menu, Load Text
- Saving code as text out to the hard disk

  File menu, Save Text

Note Visual Basic offers a useful feature for people writing code. By default it checks the syntax of your code as you are writing it. This is good news because you get constant feedback on the correctness of the syntax. Because there might be times when some people find this intrusive, this option can be disabled.

Σ String and Numeric Conversion Functions



Visual Basic provides a number of prewritten functions that make your work with strings a lot easier. Below is a partial list of prewritten functions. The focus here is on what many feel are the most important library functions. For a more detailed listing of library functions, see Table 1, "Functions, Statements, and Methods by Programming Task" in the *Microsoft Visual Basic Language Reference*.

Returns Variant	Returns String	Meaning/syntax, example, note
Chr	**Chr$**	Returns a one-character string for an ANSI code argument. For example, **Chr$**(13) + **Chr$**(10) is a carriage return and line feed, which creates as new line.
Format	**Format$**	A powerful function that displays a number in the format you request.
LCase	**LCase$**	Returns the lowercase instance of an uppercase character.
Left	**Left$**	Returns a specified number of the leftmost characters of a string. **Left$**(*stringexpression, n&*) See online Help for an example.
	Len	Returns the number of characters in a string or the number of storage bytes required by a variable.
LTrim	**LTrim$**	Returns a copy of a string with leftmost spaces removed.
Mid	**Mid$**	Returns a string that is part of another string. **Mid$**(*stringexpression$, start&, [length%]*)
Right	**Right$**	Returns a specified number of the rightmost characters in a string. **Right$**(*stringexpression, n&*)
RTrim	**RTrim$**	Returns a copy of a string with the rightmost spaces removed.

Returns Variant	Returns String	Meaning/syntax, example, note
Str	Str$	Converts a number to a string of digits.
Trim	Trim$	Removes leading and trailing spaces from a string.
UCase	UCase$	Returns the uppercase instance of a lowercase character.
	Val	Converts a string of digits to a number.

String Conversion Functions in the Employee Database

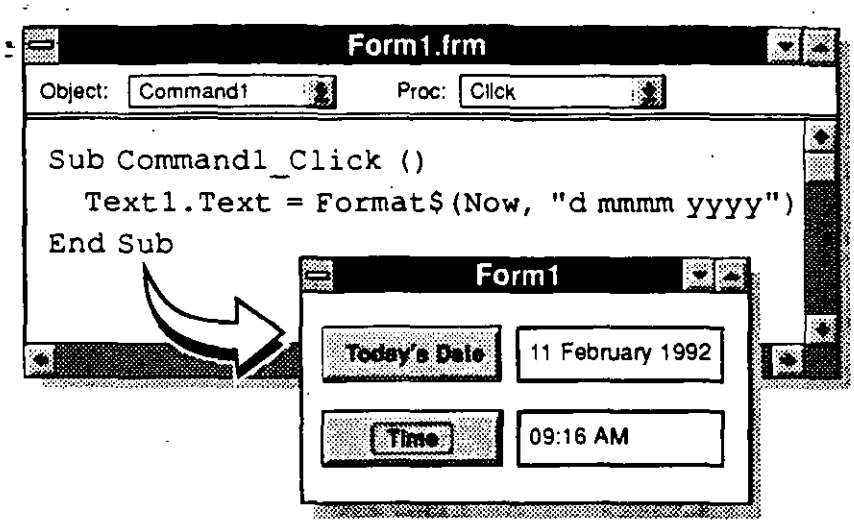
Function	Sub procedure	Form
Chr$	(not used)	
LCase$	(not used)	
Left$	(not used)	
Len	FillFields	EMPDB.FRM
LTrim$	(not used)	
Mid$	FillFields	EMPDB.FRM
Right$	cmdOK_Click	ADDPHOTO.FRM
RTrim$	FillFields	EMPDB.FRM
Str$	Form_Load	EMPDB.FRM
	FillFields	
UCase$	(not used)	
Val	cmdView_Click	EMPDB.FRM

Numeric Conversion Functions

There are a number of functions available within Visual Basic that will convert data types.

Function	Comments
CCur	Converts a numeric expression to a **Currency** value.
CDbl	Converts a numeric expression to a double-precision number.
CInt	Converts a numeric expression to an **Integer** by rounding the fractional part of the expression.
CLng	Converts a numeric expression to a **Long** (4-byte integer) by rounding the fractional part of the expression.
CSng	Converts a numeric expression to a single-precision value.
CStr	Converts a numeric expression to a **String** value.
CVar	Converts a numeric expression or **String** to a **Variant**.
CVDate	Converts an expression to a **Variant** of VarType 7 (Date).

Format$ Function



This function converts a number to a string and formats it according to instructions contained in a format expression.

Syntax **Format$**(*numeric-expression*[, *fmt$*])

fmt$

A format expression is a string of Visual Basic display-format characters that detail how the numeric expression is to be displayed.

Here are several sample format expressions and how the output would be displayed.

Format$(fmt$)	Positive 5	Negative 5	Decimal .5
0.00	5.00	-5.00	0.50
#,##0	5	-5	1
$#,##0.00;($#,##0.00)	$5.00	($5.00)	$0.50

The **Now** function can be used to return the current system date/time as a serial number. Date/time serial numbers can then be formatted with date/time or numeric formats (because date/time serial numbers are stored as floating-point values).

The following are examples of date and time formats.

Format	Display
m/d/yy	12/7/58
d-mmmm-yy	7-December-58
d-mmmm	7-December
mmm-yy	December-58
hh:mm AM/PM	08:50 PM
h:mm:ss a/p	8:50:35 p
h:mm	20:50
h:mm:ss	20:50:35
m/d/yy h:mm	12/7/58 20:50

Example

```
Sub cmdDisplayDate ()
    txtTodaysDate.Text = Format$(Now, "mm/dd/yy")
End Sub
```

A complete description of the **Format$** function can be found in Visual Basic Help.

Further Examples

Application	Form	Procedure
IconWorks	ABOUTBOX.FRM	**Form_Load**
	COLORPAL.FRM	**Txt_RGB_Change**
	COLORPAL.FRM	**Display_New_Color_and_Elements**
	ICONEDIT.FRM	**Save_Settings_To_INI_File**
	ICONEDIT.FRM	**Paste_ClipBoard_Contents**
	ICONEDIT.FRM	**Save_Colors_To_INI_File**
	ICONEDIT.FRM	**Prepare_For_New_Icon**
	ICONEDIT.FRM	**Display_Mouse_Coordinates**
	VIEWICON.FRM	**Form_Load**
	VIEWICON.FRM	**Load_All_Icons**
	VIEWICON.FRM	**File_FileList_PathChange**
	VIEWICON.FRM	**Form_Unload**

IconWorks makes extensive use of the **Format$** function. The above table is only a partial listing.

Summary

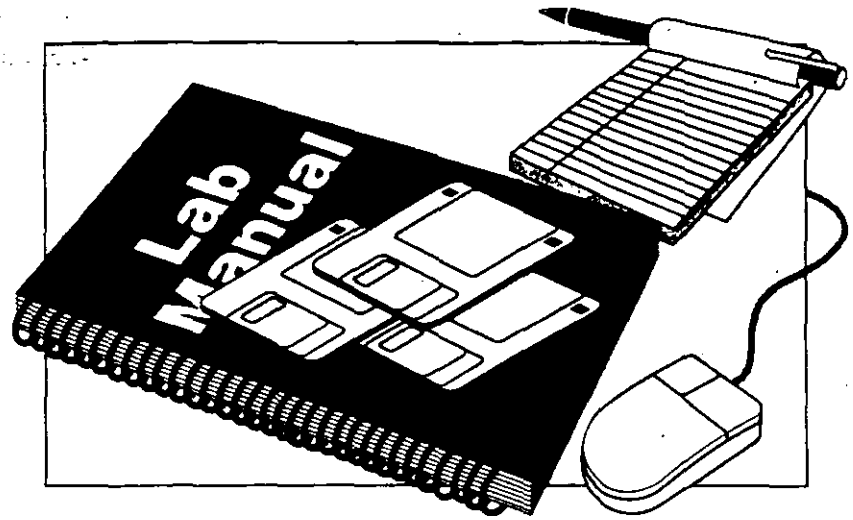
- Visual Basic Procedures
- Scope of General Procedures
- Writing Code in Visual Basic
- String and Numeric Conversion Functions

Objectives

In this module you learned to:

- Define important characteristics of **Functions** and **Sub** procedures and write code that correctly uses both.
- Define and correctly code for appropriate scope of data.
- Write code that uses the common Basic data types.

Lab Time



Go to the Writing Procedures portion of your lab manual.

Module 10: Using Conditional Logic and Loops

Overview

- **Control Structures**

  If...Then Blocks

  Select Case Statements

  Do While Loops

  Do Until Loops

  For...Next Loops

  GoTo Statements

Overview

Control structures are a crucial part of any computer language because they enable systematic decision making within the code. In this module you will learn how to control the logical flow of your program. You will also learn how to mark off blocks of code that are to be executed if a specified condition is true or false. You will also learn how to specify the number of times that a given block of statements is to be executed.

Prerequisites

This module assumes a fairly detailed understanding of coding mechanisms in Visual Basic. You will also need proficiency in designing and building the user interface for software applications. You should already be familiar with:

- Forms and properties
- Statements, **Sub** procedures, and **Function** procedures
- General and event procedures, as well as methods
- Visual Basic data types
- Variables and constants
- Use of the **MsgBox$** statement and concatenating strings
- String handling functions

Overall Objective

At the end of this module, you will be able to write code that provides systematic decision making within an application.

Learning Objectives

At the end of this module, you will be able to use:

- **If...Then...Else** blocks
- **Select Case** statements
- **Do While** loops
- **Do Until** loops
- **For...Next** loops
- **GoTo** statements

Σ Control Structures

- If...Then Blocks
- If...Then...Else Blocks
- Select Case Statements

If...Then Blocks

If *condition* **Then** *statement*

If *condition* **Then**
  statements
End If

The foil shows that there are two possible arrangements for **If...Then** blocks. You can use either a single line or multiple lines containing multiple statements. If you have multiple statements, you need to use **End If**.

Operators

There are six operators that you can use in the condition portion of the **If...Then** block.

Operator	Meaning
=	Equal
<>	Not equal
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Note In this case the = operator is *not* being used for an assignment statement, such as: `ReadOut.Caption = "0."`; instead it is being used for a conditional test.

Example

```
If optFullTime.Value = TRUE Then PositionType = "Full Time"
```

Examples in the Employee Database Code

For a list of example **If...Then** blocks, **If...Then...Else** blocks, and **Select Case** statements in the Employee Database code, see the listing at the end of this module.

If...Then...Else Blocks

```
IF condition1 Then
   statementblock1
ELSEIF condition2 Then
   statementblock2
ELSE
   statementblockn
ENDIF
```

Here you get a chance to test for several different situations and react appropriately to each one. Adding **Else** to the **If...Then** statement provides much more flexibility in the response.

Walk Through — If...Then...Else Example

∑ **To use the If...Then...Else statement**

1. From the Walk Throughs program group, start IfElse.

   When the application starts, there will be a form with a check box labeled Bold with an X in it and a command button labeled Print Something.

   The purpose of the application is to show an example of where an **If...Then...Else** block might be used.

2. Click the Print Something command button.

   The word "Something" will appear on the form in bold.

3. Click the Bold check box.

   This removes the X from the check box.

4. Click the Print Something command button.

   The word Something will appear on the form with the bold format removed.

5. Close **IfElse**.

∑ **To create the If...Then...Else example**

1. Start Visual Basic.
2. From the File menu, choose New Project.

   Start a new project.

3. Select FontBold on the Properties list box.

   View the Form property FontBold. Note that the default value is **True**. This means when the chkBold check box is created, its value should be set to Checked.

4. Double-click the check box tool in the Toolbox.

   Create a check box on the form.

5. Set the following properties for the check box:

Name	chkBold
Caption	Bold Font
Value	1 - Checked

6. Double-click the command button tool.

   Create a command button.

7. Set the following properties for the command button:

Name	cmdPrintSomething
Caption	Print Something

   Now that you have created the interface, you need to add the code to make it function.

8. From the Project window, choose View Code.
9. In the Object list box, select chkBold_Click.
10. Add the following code:

```
Const CHECKED = 1
If chkBold.Value = CHECKED Then
    Form1.FontBold = TRUE
Else
    Form1.FontBold = FALSE
End If
```

    This will cause the font to be set to bold or not, depending on how the user has set the check box.

11. In the Object list box, select cmdPrintSomething_Click.
12. Add the following:

```
Print "Something"
```

    This will print the word Something to the form. Depending on the value of the check box, it will be in bold format or not.

13. From the Run menu, choose Start.
14. From the Run menu, choose End.
15. From the File menu, choose Save As.
16. Save the form as IFELSE.FRM in \WALKTHRU\LOGIC.
17. Save the project as IFELSE.MAK in \WALKTHRU\LOGIC.

Select Case Statements

```
Select Case testexpression
   Case expressionlist1
      statementblock1
   Case expressionlist2
      statementblock2
   Case Else
      statementblockn
End Select
```

Select Case statements look a lot like **If...Then...Else** blocks, and there is a good reason for this: **Select Case** statements offer the same kind of functionality but in a much more efficient—for both the code and the coder—manner. As you can see from the example, **Select Case** statements work particularly well as a means for handling structured choices offered to the user.

Walk Through — Coding a Message Box with a Select Case Statement

In another module, you created a message box that prompted users to save their data prior to closing the application. This message box allowed users three choices: Yes, No, and Cancel.

This module has provided the tools needed to write code to detect which button the user clicked: the **Select Case** statement.

∑ To code a message box with a Select Case statement:

1. Start Visual Basic.
2. From the File menu, choose Open Project.
3. Open MSGBOX2.MAK.

   This file is located in \WALKTHRU\LOGIC and should already contain the following:

```
Sub Command1_Click ()
    ' The values for MsgBox constant declarations
    ' come from \VB\CONSTANT.TXT
    ' MsgBox parameters
    Const YESNOCANCEL = 3                ' Yes, No, & Cancel btns
    Const ICONQUESTION = 32              ' Warning query
    Msg$ = "Have you saved all your work?"
    MsgBox Msg$, YESNOCANCEL + ICONQUESTION, "MsgBox WalkThru"
End Sub
```

   In order to get the buttons to respond appropriately, you need to add a number of statements to this. Do that by declaring a variable and several constants.

4. In design mode, double-click the command button on the form.

   This opens the Code window with the event procedure code in it.

5. Add the following code to the Code window:

```
Const CANCEL = 2
Const YES = 6
Const NO = 7
Dim MB_Response As Integer
```

   The Cancel, Yes, and No declarations define more meaningful names for the values that are returned for each of the buttons. For example, if the user clicks the No button on the message box, the value 7 is returned by the system. For code readability and maintainability, you should place **Const** declarations in your code and use the constants in the **Select Case** statement. Remember, you can find the correct values to declare in \VB\CONSTANT.TXT or by searching online Help for the topic **MsgBox** function and checking the table of return values.

In the earlier message box example, you created a message box by calling the **MsgBox** statement. Even though you created a message box with three buttons, you had no way to tell which button the user clicked. Another way to create a message box is to call the **MsgBox** function. In this case, you need to declare an **Integer** variable to receive the value the **MsgBox** function returns.

Next, you need to alter the **MsgBox** statement, making it into a function call. Remember, when you call a function you must assign its return value to the appropriate type of variable. If you pass the function any arguments, the argument list must be enclosed in parentheses.

6. Edit the original **MsgBox** statement so that it reads:

```
MB_Response = MsgBox(Msg$, YESNOCANCEL+ICONQUESTION,
^"MsgBox WalkThru")
```

Now you need to add the **Select Case** statement. You may want to add only the actual code and leave the comments out.

7. Add the following code:

```
' When an integer value is returned by MsgBox function
Select Case MB_Response
    ' if it matches the value of the Const YES (6),
    Case YES
        ' indicate that the user clicked the Yes button
        Print "User clicked Yes"
    ' if it matches the value of the Const NO (7),
    Case NO
        ' indicate that the user clicked the No button
        Print "User clicked No"
    ' if it matches the value of the Const CANCEL (2),
    Case CANCEL
        ' indicate that the user clicked the Cancel button
        Print "User clicked Cancel"
End Select
```

8. From the Run menu, choose Start.

   Test the code of your application.

9. From the Run menu, choose End.

   Close the application.

   End the walk through.

Examples in the Employee Database Code

Form	Procedure/Routine	Control structure type
EMPDB.FRM	**FillFields**	**If...Then** **If...Then...Else**
	cmdDelete_Click	**Select Case** statement
	FillFields	**Select Case** statement
ADDPHOTO.FRM	**cmdOK_Click**	**If...Then...Else**

∑ Loops

- Do While
- Do Until
- For...Next

Do...Loop

Do While *condition*
  statements
Loop

– Or –

Do
  statements
Loop While *condition*

Use a **Do...Loop** to execute a block of statements an indefinite number of times. (By contrast, **For** loops let you specify how many times a set of statements are executed.) In **Do... Loop While** loops the number of times the loop is executed is controlled by a **True/False** condition. When some other event switches the value of the condition from **True** to **False**, the looping stops.

What's the Difference?

The location of the **While** condition is the key difference between the two examples of syntax above, but what does that mean? If the **While** condition is at the top of the block, the statements within the loop will not be executed if the condition is false to begin with. The block of statements will always be executed at least once if the **While** condition is placed at the bottom of the loop.

Walk Through — Coding a Do While Loop

To code a Do While loop

1. Start Visual Basic.

   From the File menu, choose New Project.

2. Double-click the command button tool in the Toolbox.

   A command button appears on the form.

3. Double-click the command button on the form.

   This will open the command button Code window.

4. Add the following code to Form1.frm:

```
Sub Command1_Click ()
    Dim I As Integer
    I = 1
    Do While I <= 5
        Print I
        I = I + 1
    Loop
End Sub
```

5. From the Run menu, choose Start.

   Run the application.

6. From the Run menu, choose End.

   End the walk through.

Do...Loop

Do...Loop Until loops are almost identical to **Do...Loop While**, but they test to see if condition is false rather than true.

Further Examples

For further examples of **...Loop While**, see the following.

Application	Module/Form	Procedure/Routine
IconWorks	ICONWRKS.BAS	**Help_File_In_Path**

Walk Through — Do...Loop While with Lists

∑ To use Do...Loop While with lists

1. From the Walk Throughs program group, start Do While with Lists.

   When the application starts, a form with a list box and a command button labeled Add To List appears.

   The purpose of the application is to show an example of using a **Do...Loop While** structure.

2. Choose the Add To List button.

   An input box will appear prompting the user to add an item to the list box.

3. Type some text and choose OK.

   The text just typed in the text box will be added as an item to the list on the loop example form.

4. Repeat the above step several times.

   Note that if you add more items to the list than can be displayed at once in the list box, a vertical scroll bar will automatically be added to allow you to view all items. In the case of this example, you must add at least seven items to the list to make a scroll bar appear.

5. Quit the application.

∑ To use Do While loops in list processing

1. Start Visual Basic.
2. From the File menu, choose New Project.

   Start a new project.

3. Set the following properties:

Caption	Do Loop While Example
Height	4035 (approximately)
Width	3870 (approximately)
Left	5565 (approximately)
Top	1290 (approximately)

4. Double-click the list box tool in the Toolbox.

   Create a list box on the form.

5. Set the following properties:

Height	1455 (approximately)
Width	1215 (approximately)
Left	960 (approximately)
Top	360 (approximately)

6. Double-click the command button tool in the Toolbox.

   Create a command button.

7. Set the following properties:

Name	cmdAddToList
Caption	Add to List
Height	375 (approximately)
Width	1215 (approximately)
Left	1080 (approximately)
Top	1920 (approximately)

   Now that you have created the interface, you need to add the code to make it function.

8. From the Project window, choose View Code.
9. In the Object list box, select cmdAddToList_Click.
10. Add the following code:

```
Dim Response As String
Do
    Response = InputBox$("Item to add:", "Add Item", "", 0, 1000)
    If Response <> "" Then List1.AddItem Response
Loop While Response <> ""
```

    By creating a **While** loop with the test at the bottom, you ensure going through the loop once to prompt for input, before any conditions are tested.

    An input box is similar to a message box, except it contains a text field for user input. The arguments for the input box are:

    Arg1 = Prompt

    Arg2 = Title bar caption

    Arg3 = Default value for text box

    Arg4 = X-position for input box

    Arg5 = Y-position for input box

    An **If** test is added so that the empty string that indicates the user wants to exit the loop will not be added as an item in the list.

    Keep this project open, because in just a moment you are going to add a **Do Until** loop.

11. Test your code.

    End the walk through.

Do Until

Do Until *condition*

statements

Loop

–Or–

Do

statements

Loop Until *condition*

Do Until is the opposite of **Do While**. The statements in a **Do Until** loop are executed only while the condition is **False**. A **Do Until** condition is the equivalent of a **Do While Not** condition.

Walk Through — Do Until Example

To use Do Until

1. From the Walk Throughs program group, start Do Until.

   This is an enhanced version of the previous Do While example. When the application starts, a form with a list box, a command button labeled Add To List, and another command button labeled Clear List appear.

   The purpose of the application is to add an example of using a **Do Until** structure.

2. Choose the Add to List button.

   An input box will appear prompting the user to add an item to the list box.

3. Type some text and choose OK.

   The text just typed into the text box will be added as an item to the list on the Loop Example form.

4. Repeat the above step several times.

   Add several items to the list.

5. Choose the Clear List button.

   All items will be cleared from the list box.

6. Quit the application.

Σ To create the Do Until example

1. Start Visual Basic.

   Make sure that you are still working on the **Do While** walk through.

2. Set the following properties for the form:

Caption	Loop Example

3. Double-click the command button tool in the Toolbox.

   Create another command button.

4. Set the following properties of the command button:

Name	cmdClearList
Caption	Clear List
Height	375 (approximately)
Width	1215 (approximately)
Left	1080 (approximately)
Top	2520 (approximately)

   Now that you have created the interface, you need to add the code to make it function.

5. From the Project window, choose View Code.
6. In the Object list box, select cmdClearList_Click.
7. Add the following code:

```
Do Until List1.ListCount = 0
   List1.RemoveItem 0
Loop
```

   A **Do Until** loop can be used to keep removing items from List1 until the list box is empty. The ListCount property keeps track of the number of items currently in the list.

   Inside the loop is a statement to call the **RemoveItem** method. When you pass a zero to **RemoveItem**, you are telling it to remove the top item in the list.

   Rather than setting up a loop, you can clear an entire list box with the **clear** method.

```
List1.clear
```

8. From the File menu, choose Save File As.

   Save this file as DOUNTIL.FRM and the project as DOUNTIL.MAK in the \WALKTHRU\LOGIC subdirectory. You will need these files in the For Loop lab.

   End the walk through.

For...Next

For *counter* = *start* **To** *end* (**Step** *increment*)
(*statements*)
(**Exit For**)
(*statements*)
Next (*counter*)

For...Next loops are used to execute a block of statements a fixed number of times. The key difference between a **For...Next** loop and a **Do...Loop** is that a **For...Next** loop includes a counter that increases or decreases with each repetition of the loop.

Complete the demonstration for an example of **For...Next** loops.

Walk Through — Using Visual Basic For...Next Loops

To use For...Next loops

1. Start Visual Basic.

   Open Visual Basic. Open a new form.

2. From the File menu, choose New Project.
3. Double-click the command button tool in the Toolbox.

   This will open the command button Code window.

4. Add the following code in the Code window:
5. Add the appropriate code to the Command1_Click event:

```
Sub Command1_Click ()
    Dim I As Integer
    For I = 1 to 6
        Print I
    Next I
End Sub
```

6. From the Run menu, choose Start.

   Test the application you have created. The default value for Step is 1. The application prints the numbers 1 through 6 on the form.

7. From the Run menu, choose End.
8. In the Code window, change the Step value to 2.
9. From the Run menu, choose Start.

   Test the application again.

   How has the output you are getting changed?

   How would you change the **For...Next** loop to display only even numbers?
10. From the Run menu, choose End.

    End the walk through.

A more sophisticated use of the **For...Next** loop can be found in Visual Basic Help.

Example

```
'For...Next Statement Example
Sub ForNextDemo ()
    NL$ = Chr$(13) + Chr$(10)          ' Define newline.
    For Rep% = 5 To 1 Step -1          ' Set up five repetitions.
            ' Equate alphabet to numbers.
        For Indx% = Asc("A") To Asc("Z")
                ' Append each letter to string.
            Msg$ = Msg$ + Chr$(Indx%)
        Next Indx%
        Msg$ = Msg$ + NL$              ' Add newline for each rep.
    Next Rep%
    MsgBox Msg$                        ' Display results.
End Sub
```

Notice that one **For...Next** loop is "nested" inside the other in this example? This is a powerful tool for processing multiple rows or columns in arrays.

If you want to see this code running, follow the directions detailed above and make the appropriate modifications to the code so that it executes as the result of a command button being clicked.

Walk Through — Another For...Next Loop Example

∑ To use another For...Next loop

1. From the Walk Throughs program group, start ForLoop to Clear List.

   This is an enhanced version of the previous **Do While** example. When the application starts, a form with a list box, a command button labeled Add To List, and another command button labeled Clear List appear.

   The purpose of the application is to change the example to use a **For...Next** loop to clear the list.

2. Choose the Add to List button.

   An input box will appear prompting the user to add an item to the list box.

3. Type some text and choose OK.

   The text just typed into the text box will be added as an item to the list on the loop example form.

4. Repeat the above step several times.

   Add several items to the list.

5. Choose the Clear List button.

   All items will be cleared from the list box.

6. Quit the application.

∑ To create the For...Next loop example

1. Start Visual Basic.
2. From the File menu, choose Open Project.
3. Open DOUNTIL.MAK.

   Open the project from the **Do Until** Walk Through. It is located in the \WALKTHRU\LOGIC subdirectory.

4. From the Project window, choose View Code.
5. In the Object list box, select cmdClearList_Click.
6. Replace the existing code with:

```
Dim I As Integer
For I = 0 To List1.ListCount-1
   List1.RemoveItem 0
Next I
```

   You can use a **For...Next** loop to keep removing items from List1 until the list box is empty. In this case, the range for the loop will be from 0 (the top item in the list) to ListCount–1 (the last item in the list).

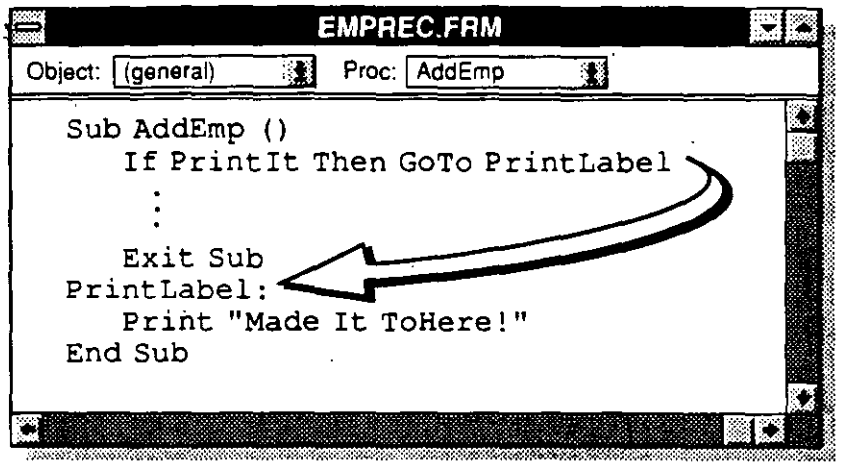
   Inside the loop is a statement to call the **RemoveItem** method. When you pass a zero to **RemoveItem**, you are telling it to remove the top item in the list.

Further Examples

For further examples of a **For...Next** loop see:

Application	Module/Form	Procedure/Routine
Employee Database	EMPDB.FRM	Form_Load

The GoTo Statement



GoTo causes execution to jump from the **GoTo** statement to another location marked by a label or line number.

Even though the **GoTo** statement doesn't enjoy much favor these days, Visual Basic supports it, and there are several places where there is an obvious use for it. Error handling is a particularly good example.

Note **GoTo**s are not like procedures; there is no standard return from a **GoTo** as there is from a procedure.

Style Guidelines

- Each line label must begin with an alphabetic character.
- Each line label must end with a colon.
- Each line label must be unique within its own module.
- Each line label can have no more than 40 characters.
- Do not use Visual Basic keywords in line labels.
- Line labels are not case sensitive.
- A line label must start with the first nonblank character on a line, but it need not be in the first column. Visual Basic forces it to the leftmost column.

Examples

Additional sample code is located in Visual Basic Help.

For further examples of **GoTo** statements see the following.

Further Examples

Application	Module/Form	Procedure/Routine
IconWorks	ICONWRKS.BAS	**Validate_FileSpec**
Employee Database	EMPREC.FRM	**AddEmp**

Summary

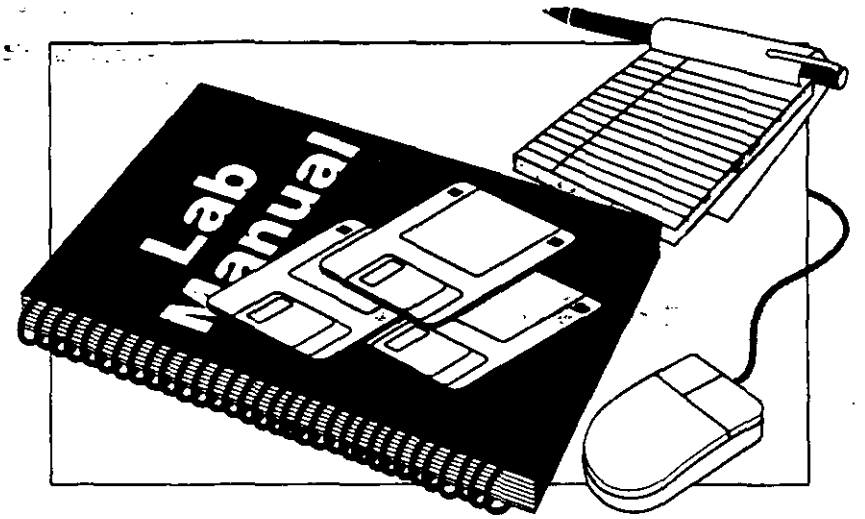
- **Control Structures**
  - **If...Then** Blocks
  - **Select Case** Statements
  - **Do While** Loops
  - **Do Until** Loops
  - **For...Next** Loops
  - **GoTo** Statements

Objectives

In this module you learned to use:

- **If...Then...Else** blocks
- **Select Case** statements
- **Do While** loops
- **Do Until** loops
- **For...Next** loops
- **GoTo** statements

Lab Time



Go to the Conditional Logic and Loops portion of your lab manual.

Module 11: Debugging Code in Visual Basic

Σ Overview

- **Debugging Terms**
- **Debugging Code in Visual Basic**

  Using the Call Tree

  Using Watch Variables to Monitor Program Execution

Overview

Visual Basic offers programmers of all skill levels a set of robust tools to use during application development.

Prerequisites

Prior to starting this module, you should already be familiar with:

- Controls, forms, and properties
- **Function** and **Sub** procedures
- General and event procedures

Overall Objective

The overall objective of thi odule is to introduce you to some of the most useful debugging tools available isual Basic. This is not intended to be comprehensive treatment of debugging techniques in general.

Learning Objectives

At the end of this module, you will be able to:

- Distinguish among run, design, and debug modes in Visual Basic.
- Use the Watch window to display the current values of variables within a program.
- Set breakpoints within code.
- Single step through application procedures.

Debugging Terms

- **Watch Expressions**
- **Watch Point Variables**

  Watch point — break when true

  Watch point — break when changed
- **The Debug Window**

  The Immediate pane (?, =)

  Watch pane

Watch Expressions

A watch expression is a variable whose value is displayed in the Debug window whenever a program enters break mode.

You can set a watch expression by opening the Debug menu and choosing either the Add Watch or Edit Watch command. This can be done in either design mode or break mode.

Watch Point Variables

A watch point variable will cause a program to enter break mode whenever the watch point condition is satisfied.

You can set a watch point variable by opening the Debug menu and choosing either the Add Watch or Edit Watch command. This can be done in either design mode or break mode.

Watch Point — Break When True

This watch point variable will cause a program to enter break mode whenever the variable's value becomes true.

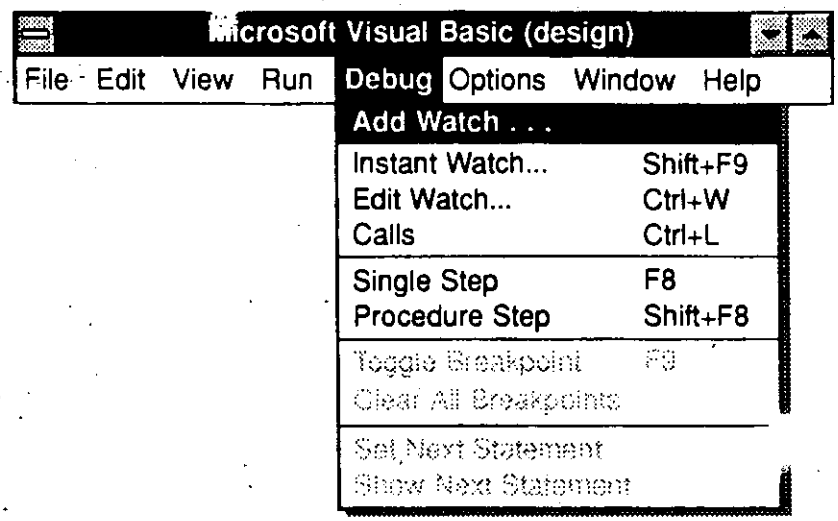
Watch Point — Break When Changed

This watch point variable will cause a program to enter break mode whenever the variable's value changes.

The Debug Window

The Debug window is broken into two parts: the Immediate pane (the lower half of the window) and the Watch pane (the upper half). The Immediate pane is where you can have an interactive conversation with the debugger using either the question mark (?) or the equal sign (=). The Watch pane is where the watch variables are displayed during break mode.

Debugging Code in Visual Basic



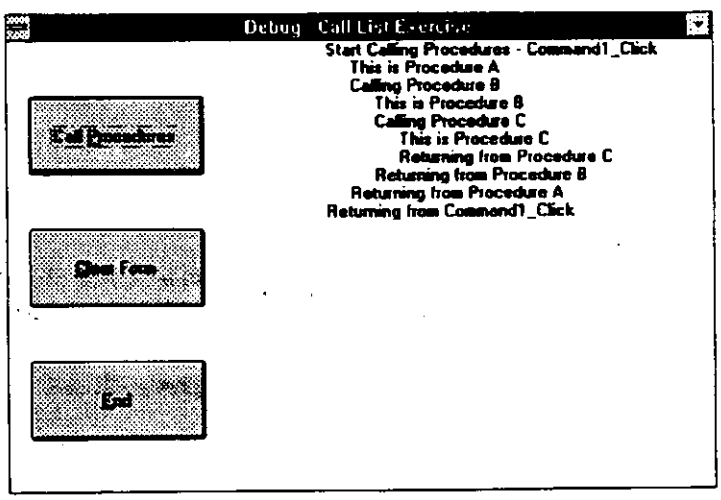
Using the Visual Basic Debugger

This module is actually made up of a number of walk throughs that introduce you to the various tools available within the Visual Basic debugger.

Walk Through — Using the Call Tree

To see the first sample application work

1. From the Walk Through program group, start Debug1.
2. Click the Call Procedures command button.
3. Observe the output generated on the form.



Note that Command1_Click calls ProcedureA, ProcedureA calls ProcedureB, and then ProcedureB calls ProcedureC.

4. Choose End.

Σ To examine the code for the walk through

The purpose of this walk through is to introduce you to two tools that you will find very useful: the Call window and single stepping through code.

1. Start Visual Basic.
2. Open the DEBUG1.MAK file located in \WALKTHRU\DEBUG.
3. Select MOD1.BAS in the Project window and click View Code.
4. From the Procedures drop-down list box, select ProcC.
5. Examine the code for ProcC.

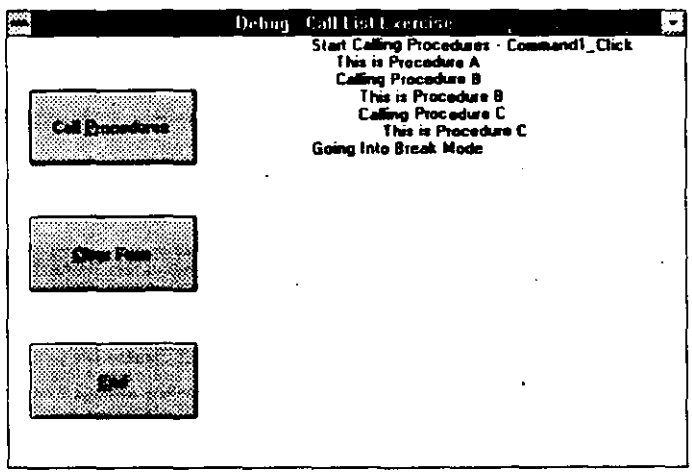
   As it stands, this procedure prints two statements at column 49 on your form. The other three lines are "commented out."

6. Remove the three comment marks in front of the **Stop** and **Print** statements.

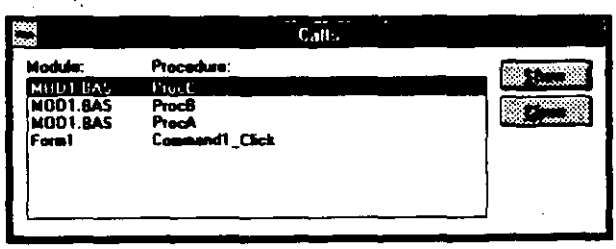
   What will happen when you do this? You have added three new statements to the application. The first one prints a message to the form at column 40 that tells you the application is going into break mode. The second statement actually stops that application using **Stop**. The final statement prints a message to the form that tells you that you have entered single-step mode after pressing F8 twice.

7. From the Run menu, choose Start.
8. Click the Call Procedures command button.

   If you move the Code window from on top of Form1, you will see that the output is slightly changed. Now it should look like this.



9. From the Debug menu, choose Calls. Now you should see a form like this.



Notice that the calls are listed with the most recently called at the top.

10. Click the Show command button.

    This returns you to MOD1.BAS.

    You should see the code for ProcC with a box around the code line containing the **Stop** statement.

11. Single step by pressing F8 once.

    This will start you single stepping through the rest of the procedure. This makes `Print "Going into Single Step Mode"` the next statement to execute.

    If you position both windows so that you can see almost all of them, you will see Visual Basic work as you single step through the application.

12. Press F8 one more time.

    This executes the current **Print** statement and makes `Print "Returning from Procedure C"` the next statement to execute.

13. Press F8 a third time.

    This executes the current **Print** statement (check the output on the form) and makes the **End Sub** statement the next statement to execute.

14. Press F8 a fourth time.

    This executes the **End Sub** of ProcC and takes you to the next statement to execute in ProcB, which is the `Print "Returning from Procedure B"` statement.

15. From the Debug menu, choose Calls or press CTRL+L.

    Note the most recent procedure is now Procedure B.

    Doing this verifies that the call list has changed since the return from ProcC.

16. Click the Show command button to return to ProcB.

17. Press F8 a fifth time and then a sixth time.

    This takes you through the remaining two Procedure B statements and takes you to the next statement in ProcA—`Print "Returning from Procedure A"`.

18. Press F8 twice more.

    This single steps you through the remaining Procedure A statements and takes you to the next statement in Command1_Click—`Print "Returning from Command1_Click"`.

19. Press F8 twice more.

    This takes you through the remaining the Command1_Click event procedure.

20. From the Run menu, choose Break.

21. From the Debug menu, choose Calls.

    Notice that *no calls* are listed. That is because no **Sub** or **Function** procedure is currently "open" and your application is in idle waiting for the user to do something.

22. Choose the Close command button on the Calls dialog box.

23. From the File menu of Visual Basic, choose Exit.

    If you want to try this exercise a second time, do not save the changes to the files.

Walk Through — Using Watch Variables to Monitor Program Execution

Debugging Applications

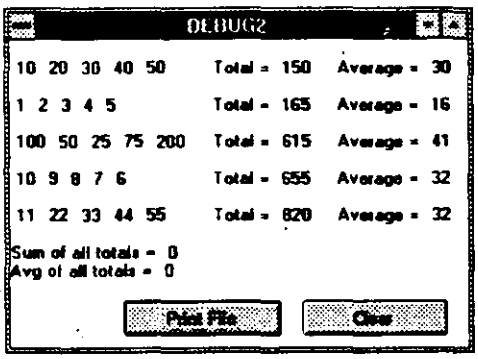
Debugging applications is somewhere between a science and an art. By careful use of your debugging tools, such as breakpoints, watch variables, single stepping, and procedure stepping, you can zero in on the logical bugs existing in your code.

This walk through applies breakpoints and watch expressions to locate a number of logical errors contained within the code.

To see the second debugging application run

1. From the Walk Through program group, start Debug2.
2. Click the Print File command button.

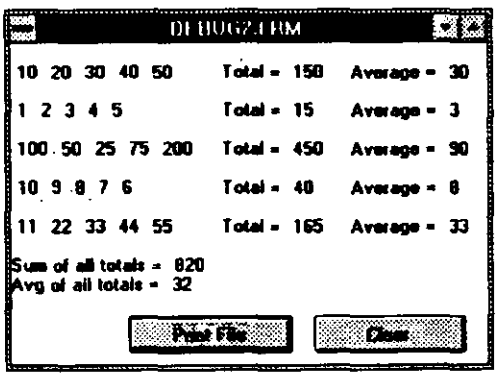
   You get output that looks like this.



   This program lists five numbers, their total, and their average. It does this for five sets of data. At the end, it prints the total of all the numbers and the average for all the numbers.

3. But examine the output carefully. It isn't giving you what you want: The totals and averages aren't correct.

   If the application were coded correctly, final output should look something like this.



   You need to correct the logic, but where do you begin?

4. Double-click the Control menu on the Debug2 form to close the application.

∑ To analyze the variables used in the sample application

1. If Visual Basic is not running already, start it.
2. From the File menu, choose Open Project to locate and start DEBUG2.MAK, located in \WALKTHRU\DEBUG.
3. Note that there are two files to this project: DEBUG2.FRM and MOD2.BAS.
4. What are the *global-level* variables, and where they are located?

If you inspect the General Declarations section of MOD2.BAS, you will see `x(5)`, an **Integer** array, and sumcounter declared as an **Integer**.

There is also a **String** variable called FilenameS in MOD2.BAS.

5. What are the *form-level* variables, and where they are located?

If you inspect the General Declarations section of DEBUG2.FRM, you will find Sum declared as an **Integer**. But, be careful here. Visually inspecting the General Declarations section doesn't tell you all you need to know, and this will come back to haunt you a little later on.

Why do you need to know this? Awareness of the types and scope of the variables within your application will help in the analysis of the code.

∑ To analyze the procedures used in the sample application

1. In DEBUG2.FRM, examine the cmdPrintFile_Click event procedure.
2. Which general procedure is called in this procedure?

If you said the Readfile procedure from MOD2.BAS, you were correct.

3. Examine the ReadFile general procedure in MOD2.BAS.

   To display the source for the ReadFile general procedure, place the cursor in the word ReadFile in the cmdPrintFile_Click procedure and press SHIFT + F2

   Note the outer **For...Next** loop is controlled by J, whereas the inner **For...Next** loop is controlled by I. These two loops control part of the calculations in your application.

∑ To use breakpoints and watch expressions to observe program behavior

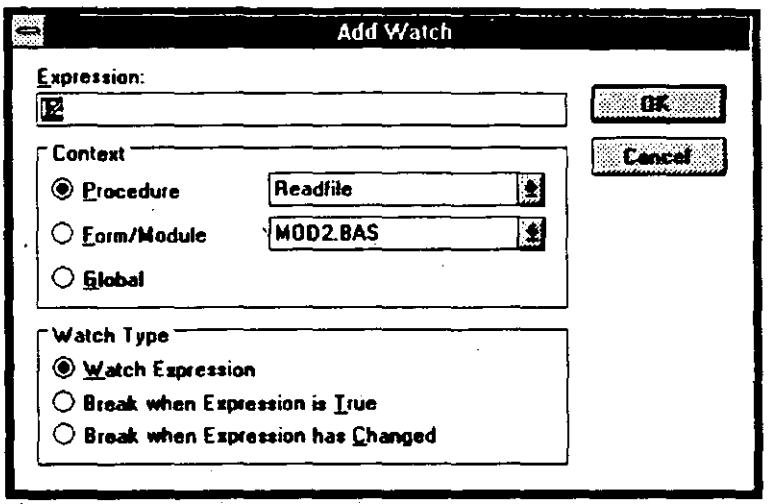
In order to observe the logic flow and I and J's values, we will use two watch expressions and a breakpoint.

1. Set a breakpoint on the `Next I%` line.

   Place the cursor anywhere on the `Next I%` line and press F9.

2. Highlight the I% variable on the `For I%` line.
3. From the Debug Menu, choose Add Watch.

   A form will appear on screen that looks like this.



4. Make sure that the values for the Add Watch window are as follows.

Control	Setting
Expression	I%
Procedure	Readfile
Form/Module	MOD2.BAS
Watch Type	Watch Expression

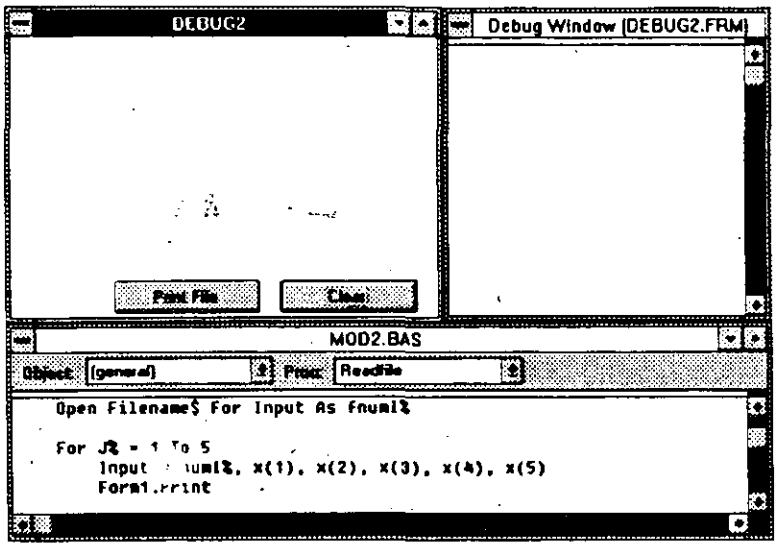
5. Choose OK.
6. Repeat the above steps (1 thorough 5) for J% to create a watch expression for J%.

   You now have two watch expressions. Each time your program enters into break mode, the current values of these two variables will be displayed at the top of the Debug window.

Σ To observe your watch expressions in action

1. Start your program by pressing F5.
2. Arrange the windows so that you can see DEBUG2.FRM, the Debug window, and MOD2.BAS code window.

   You might want to arrange them so that they look like this.



3. Choose the Print File button.

   Because you have set a breakpoint, your application will stop execution on the line `Next I%`.

4. Note the values of I% and J% in the Debug window.

   Right now they should each be at 1 because you are in the first iteration of both loops. If you look at the output of DEBUG2.FRM, you will see that the value `10` has been printed there.

5. Continue execution by repeatedly pressing F5 until J% equals 2 and I% equals 5.

   Each time you press F5, your application enters break mode and the current values of I% and J% are displayed in the Watch pane.

   At this point you know that I% and J% are behaving correctly. It is time to pursue analysis in a different direction.

6. From the Run menu, choose End to stop execution of your program.
7. Press F9 to clear your breakpoint on the `Next I%` line.

   The `Next I%` line in the Readfile procedure of MOD2.BAS should already be selected; so press F9.

To this point, you have seen what we wanted to demonstrate about watch points and watch expressions. If that is all you need from this exercise, then quit here. If, however, you want to sharpen your debugging skills, continue on with the exercise.

The question here is this: What is going wrong with the application? First, it doesn't seem to be calculating the individual totals properly. You are ending up with a value of 165 for the second set of numbers when it should be 15. Why don't you next set a watch point on Total%?

Σ To use watch points—break when expression has changed (Total%)

1. In the Code window for MOD2.BAS, locate the procedure Readfile and highlight the variable Total% within the For I% loop.
2. From the Debug menu, choose Add Watch.
3. Make sure that all the controls for the watch point are as follows.

Control	Setting
Expression	Total%
Procedure	Readfile
Form/Module	MOD2.BAS
Watch Type	Break when Expression has Changed

4. Choose OK to close the Add window.

Σ To observe your watch variable in action

1. From the Run menu, choose Start to run your program.

   Note the entry added to the top of the Debug window. This will cause the program to enter into break mode whenever Total% changes.

   You may need to expand the Watch pane display area in order to see the additional watch variable.
2. Click the Print File command button to enter the Readfile procedure.
3. What is the first value for Total%?

   _____

   If you said 10, you were correct.
4. Press F5 six more times (so that I% equals 1 and J% equals 2).
5. Now, what is the value of Total%?

   _____

   If you said 151, you were correct; but what should the value of Total% be at this point, and what does that tell you?

   _____

   The value for Total% at this point should be 1, because that is the first value for the second pass through the J% loop. What this tells you is that you need to reset the value of Total% before each entry into the I% loop.
6. From the Run menu, choose End.
7. Add the following line just before the `For I% = 1 to 5` line:

```
Total% = 0
```

8. Test your change by running the code again. Notice that the total is correct.

   You will notice that the averages for the last four sets of numbers have the values 1, 30, 2, and 6. That is, they are still wrong. To fix this problem, what other variable must you zero out before reentering the I% loop?

   If you said counter%, you were correct.

9. From the Run menu, choose End.
10. Add the following line just before the `For I% = 1 to 5` line:

```
Counter% = 0
```

11. Test by running the program again.
12. Inspect the Total and Average for each of the five sets of data.

    Total and Average for each set of numbers should now be correct. Compare them with the correct output displayed earlier in this walk through.

13. Inspect the values displayed for the overall sum and average values.

    They are both 0. That can't be right.

14. From the Run menu, choose End.

∑ To clear all watch variables and all breakpoints

1. From the Debug menu, choose Edit Watch.
2. Choose Delete All to delete all watch points you may have set.
3. Choose Close to close the Edit Watch dialog box.
4. From the Debug menu, choose Clear All to clear all breakpoints that are set.

∑ To use watch points—break when expression becomes True (Sum)

At this point you have fixed the calculation of individual totals and averages, but the calculation of the overall total and average is still wrong. For this reason, closer examination of the Sum variable is warranted.

1. From the Debug menu, choose Add Watch.
2. Fill in the following entries:

Expression	Sum > 0
Form/Mod	DEBUG2.FRM
Procedure	cmdClear_Click

   Break when Expression is True

   Choose OK

3. From the Run menu, choose Start.
4. Note the entry added to the top of the Debug window.

   This will cause the program to enter into break mode whenever Sum > 0 becomes **True**.

∑ To observe your watch variable in action

1. Press the Print File command button to run the program.
2. Did you ever enter Break mode?

Your program should not have entered break mode because the variable Sum was never greater than 0.

3. From the Run menu, choose Break. Notice in the debug window it specifies Sum <Not in Context>. This is because Sum is a local variable and you are not currently running the procedure that contains Sum.
4. From the Run menu, choose End.

∑ To fix the code—using debugging and logical analysis to correct the code

1. First of all, how many variables named Sum are in this application?

Remember at the start of this walk through that we asked that question and said that there was only one. Well, there are really two. If you said two, you were correct.

2. Look in the General Declarations section of DEBUG2.FRM. Notice that it contains an explicitly declared *form-level* integer variable called Sum. This variable is not available outside this form.
3. Now look at the code in MOD2.BAS. Is the variable called Sum there, the same one as in DEBUG2.FRM? No, this is an implicit variable local to the **Sub** procedure Readfile. This is the source of your problem.

   How can you make sure that the variable called Sum in cmdPrintFile in DEBUG2.FRM and the one referred to in Readfile are the same?

If you said delete the local variable declaration and make it Global, you were right.

4. Delete the declaration in DEBUG2.FRM.
5. Add the following declaration to MODULE2.BAS:

```
Global Sum As Integer
```

∑ To test your scoping changes

1. Run the program again.

   After making these changes, you'll discover that the sum of all totals is 2025 and average of all totals is 81. They're still not right.

2. Choose End from the Run menu.

3. Set the breakpoint on Next I% and the watch expressions on I% and J% in the Readfile procedure. (Follow the guidelines outlined earlier in the walk-through, if you need to.)

Σ To single step through your program

1. From the Run menu, choose Start.
2. Click the Print File command button.

   The next line of code to be executed should be `Next I%`.
3. Now, query for the current value of Sum by placing your cursor in the Debug window and typing:

   ?Sum

   Then press the ENTER key.
4. Observe that the value 10 displayed in the Immediate pane of the Debug window. That's the current value of Sum.
5. Step through the I% loop again, stopping at the same point as before, and query the value of Sum.
6. Observe that the value 40 is displayed. However, 10 + 20 does not equal 40.

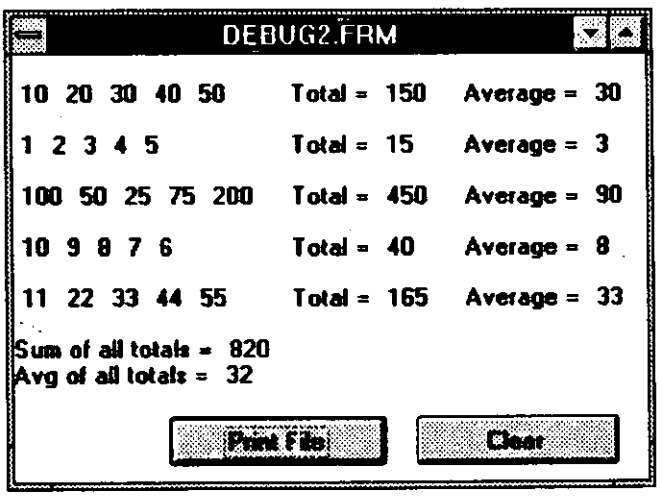
   What's the problem?

If you said the line: `Sum% = Total + Sum` needs to be moved below the line: `Next I%`, you're correct.

7. Cut and paste the line `Sum% = Total + Sum` to just below `Next I%`.

Σ To test your logic changes

1. Start and test the application again to see if it's working properly now.
2. Compare your output with the desired final output shown below.



3. Close the application and give yourself a pat on the back.

Summary

- **Debugging Terms**
- **Debugging Code in Visual Basic**

  Using the Call Tree

  Using Watch Variables to Monitor Program Execution

Objectives

In this module you learned to:

- Distinguish among run, design, and debug modes in Visual Basic.
- Use the Watch window to display the current values of variables within a program.
- Set breakpoints within code.
- Single step through application procedures.

Module 12: Printing to Forms and Printers

Σ Overview

- Scenario
- Methods for Printing
- Print-Related Functions
- Using PrintForm

Overview

Printing to forms and printers can be as complicated or as simple as you want to make it. This module shows you some of the simple tricks that you can use to prepare text for printing.

Prerequisites

To succeed in the module, you should already be familiar with:

- The general syntax for methods.
- **Select Case** statements and **If...Then...Else** blocks.

Overall Objectives

The purpose of this module is to give you an introduction to printing using Visual Basic. This module is split into two topics: printing to a form and sending that form to a printer.

Learning Objectives

At the end of this module, you will be able to:

- Use the **Print** method to create output directly onto a form.
- Describe how the AutoRedraw property relates to printing directly to a form.
- Use the CurrentX and CurrentY properties of a form and the **Spc** and **Tab** functions to control the placement of output printed to a form.
- Use the **Cls** method to clear a form.
- Use the **PrintForm** method to send a bit-for-bit image of a form to the printer.

Scenario

- **Printing a Form**

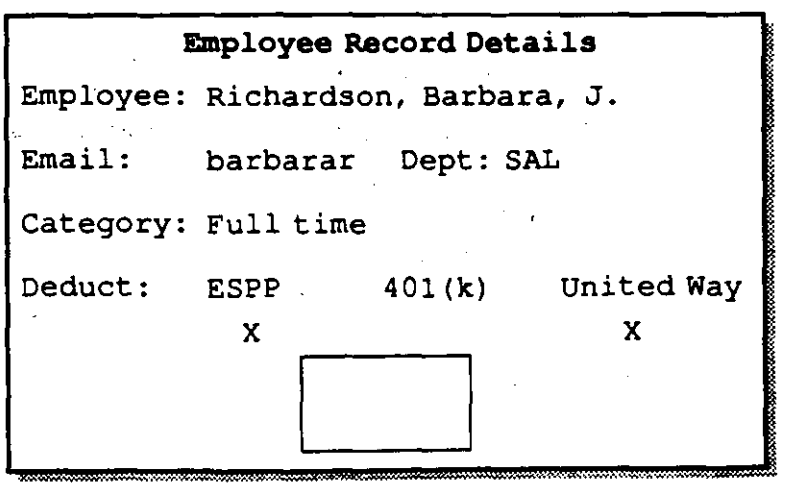
The specification for the Employee Database requires that users be able to print out an individual employee's personnel information.

Individual Reports

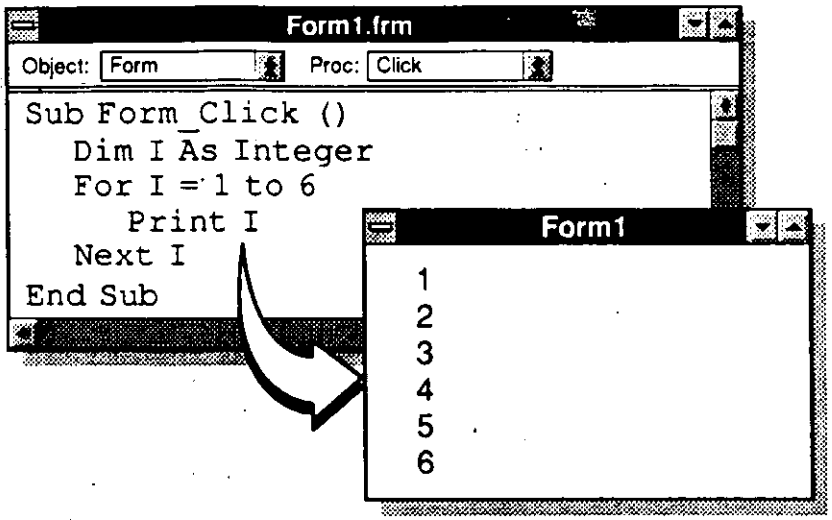
You have determined that users want the individual report to include the following information on individual lines:

1. Last name, first name, and middle initial
2. Electronic mail alias and department code
3. Position type
4. Deduction information
5. Employee photo centered at the bottom of the page

After much work, you have determined that a workable individual report form would look like this.



Methods for Printing



Overview

There are a number of different methods, functions, and properties you will need to understand in order to print to a form and then send that form to a printer.

Methods	Functions	Properties
Print	**Spc**	CurrentX
Cls	**Tab**	CurrentY
PrintForm	**Format**	AutoRedraw, Fontsize, and so on

The Printer Object

The Printer object is a predefined object in Visual Basic. You can send output to the Printer object using the **Print** method. When you are finished placing information on the Printer object, you use the **EndDoc** method to send the output to the printer. The output will print on your default printer.

```
Printer.Print       "Here is some information on Page one."
Printer.NewPage     'This causes a page break
Printer.Print       "Here is some more information"
Printer.EndDoc      'This sends the output to the printer.
```

The Print Method

The **Print** method is used to put a text string on a form, a picture box, or Printer object using the current color and font. This portion of the module will discuss the **Print** method by first talking about it in general and then discussing the implications of printing to a form and printing to a picture box.

Syntax

[*object.*]**Print**[*expressionlist*][{;|,}]

The object can be either a form, a picture, or a printer to which the expression list will be printed.

The expression list is either text or numbers that you want to print. Multiple expressions can be separated with a semicolon, a comma, or a space. If the expression list is omitted, Visual Basic prints a blank line.

The semicolon and the comma are used to specify the location of the text cursor for the next character displayed. A semicolon means the cursor is placed immediately after the last character displayed. The comma means the cursor is placed at the start of the next print zone. Print zones begin every 14 columns.

Note A form does not need to have the Visible property set to **True** to be able to send output to it.

Walk Through — Printing with For...Next Loops

To print with For...Next loops

1. From the Walk Throughs program group, start Fancy Print.
2. Click the form.

   This displays the output from the code contained on the next page.
3. From the Control menu, choose Close.

The following example is available in Visual Basic Help.

Example

```
'Print Method Example
Sub PrintDemo ()
    Const BLUE = 1
    Const MAGENTA = 5
    Const BRIGHTWHITE = 15
    Cls                                 ' Clear form.
    Width = 7200                        ' Set form width.
    Height = 5000                       ' Set form height.
    BackColor = QBColor(BLUE)           ' Set background color.
    For I% = 1 To 3
        Select Case I%
            Case 1                      ' First time.
                'Set foreground color.
                ForeColor = QBColor(MAGENTA)
                K% = 1: L% = 9: M% = 1
                Msg$ = "Visual"         ' Set message."
                CX = 0                  ' Set position variable.
            Case 2                      ' Second time.
                K% = 1: L% = 9: M% = 1
                Msg$ = "Basic"
                CX = ScaleWidth
            Case 3                      ' Third time.
                ForeColor = QBColor(BRIGHTWHITE)
                K% = 9: L% = 1: M% = -1
                Msg$ = "Visual Basic"
                CX = ScaleWidth / 2
        End Select
```

```
28        For J% = K% To L% Step M%
29           Select Case J%                    'Change font size.
30              Case 1: Fontsize = 8
31              Case 2: Fontsize = 10
32              Case 3: Fontsize = 12
33              Case 4: Fontsize = 14
34              Case 5: Fontsize = 18
35              Case 6: Fontsize = 20
36              Case 7: Fontsize = 24
37              Case 8: Fontsize = 36
38              Case 9: Fontsize = 48
39           End Select
40           If I% = 1 Then
41              Offset% = 0                    ' Text on left.
42           ElseIf I% = 2 Then
43              Offset% = TextWidth(Msg$)      ' Text on right.
44           Else
45              ' Text in center.
46              Offset% = TextWidth(Msg$) / 2
47           End If
48           CurrentX = CX - Offset%
49           Print Msg$                        ' Print message.
50        Next J%
51        CurrentY = 0                         ' Reset to top of form.
52     Next I%
53  End Sub
```

For further examples using the **Print** method, see the following.

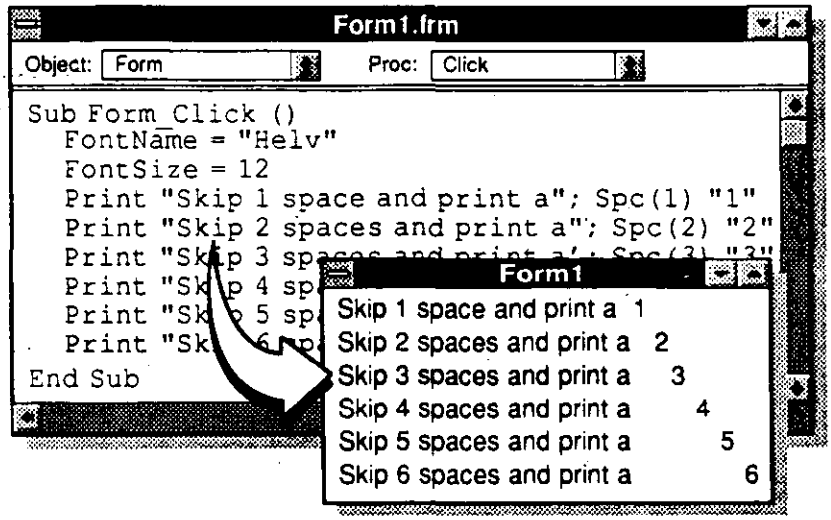
Further Examples

Application	Form	Procedure
IconWorks	ICONEDIT.FRM	**DisplayMouseCoordinates**
	ICONEDIT.FRM	**Pic_StatusArea_Paint**
	VIEWICON.FRM	**Load_All_Icons**
	VIEWICON.FRM	**Pic_SelectedIconLa el_Paint**
Employee Database	EMPREC.FRM	**cmdAddDeleteUpdate_Click**

Σ Print-Related Functions

- Spc Function
- Tab Function

Spc Function



This function inserts a specified number of blank characters in a **Print** method, starting at the current print position.

Syntax

Spc(*number%*)

The *number* argument must be an integer between 0 and 32,767 inclusive.

Important When using fixed pitch fonts (where the space allowed for each character is the same size), **Spc** is not complicated to use; but with a proportionally spaced font (such as Times New Roman®), the width of the space is always the average width of all characters in the point size of that font.

Example

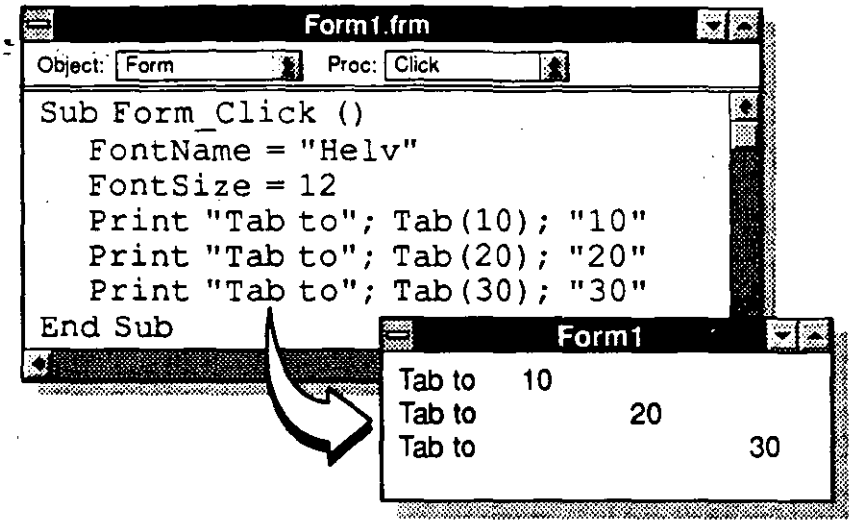
```
1   'Spc Function Example
2   Sub Form_Click ()
3   FontName = "Courier"
4   Print "         1         2         3         4         5"
5   Print "12345678901234567890123456789012345678901234567890"
6   Print "I'll skip some spaces then print an x"; Spc(2); "x"
7   End Sub
```

For other examples using the **Spc** function, see the following.

Further Examples

Application	Form	Procedure
Employee Database	EMPREC.FRM	**cmdAddPrintUpdate_Click**

Tab Function



The **Tab** function moves the text cursor to a specified print position when used with the **Print** method.

Syntax

Tab(*column%*)

The parameter *column%* is an **Integer** expression that is the column number of the new print position. The leftmost print position on an output line is always 1.

For forms, the only limit to the rightmost print position is the range of the **Integer** data type.

For a complete description of the **Tab** function, see Visual Basic Help.

Using PrintForm

(*form.*)**PrintForm**

PrintForm

Once you have finished formatting all the output to a form, you can use the **PrintForm** method to send a bit-by-bit image of the form to a printer.

Syntax

[*form.*]**PrintForm**

Example

This example is available in Visual Basic Help. If your machine is correctly connected to a printer, this code will send the current form to it.

```
'PrintForm Method Example
Sub PrintFormDemo ()
    On Error GoTo ErrorHandler        ' Set up error handler.
    PrintForm                         ' Print form.
    Exit Sub
ErrorHandler:
    Msg$ = "The form could not be printed. "
    MsgBox Msg$                       ' Display message.
    Resume Next
End Sub
```

Further Examples

Application	Form	Procedure
Employee Database	EMPREC.FRM	**cmdAddPrintUpdate_Click**

Remember, the **Print** method is used to place the data on the form. The **PrintForm** method is used to send a bit-by-bit image of the form to the default printer.

You should set the AutoRedraw property of a form to **True** if you will print it using the **PrintForm** method. If AutoRedraw is not **True**, graphics drawn directly on the form using **Print**, **Line**, **Circle** and other graphic statements will not appear on the printer.

Walk Through — Effects of AutoRedraw

Σ To see the effects of setting the AutoRedraw Property

1. From the Walk Throughs program group, start PrintForm & AutoRedraw.
2. Click the Draw command button.

   This draws lines on the form.
3. From the AutoRedraw menu notice that AutoRedraw is set to False.
4. Drag the minimized Program Manager icon onto the form and then move it away.

   Notice the form now has a gap where the Program Manager icon erased the lines. The lines are not redrawn. If you had printed this form using the **PrintForm** method, the lines would not appear.
5. From the AutoRedraw menu choose True to set AutoRedraw to True.

   The code in this menu selection invokes the Draw button for you.
6. Drag the minimized Program Manager icon onto the form and then move it away.

   Notice the form automatically redraws the lines if necessary. If you had printed this form using the **PrintForm** method, the lines would appear.

Summary

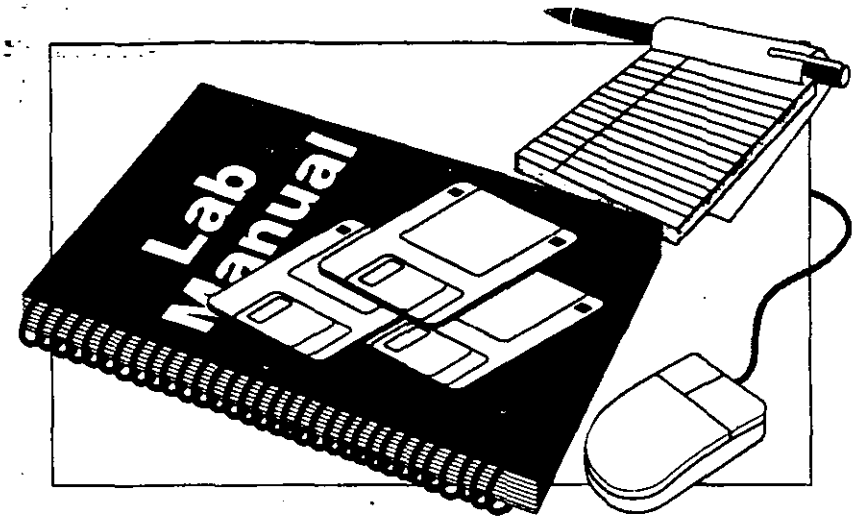
- Scenario
- Methods for Printing
- Print-Related Functions
- Using PrintForm

Objectives

In this module you learned to:

- Use the **Print** method to create output directly onto a form.
- Describe how the AutoRedraw property relates to printing directly to a form.
- Use the CurrentX and CurrentY properties of a form and the **Spc** and **Tab** functions to control the placement of output printed to a form.
- Use the **Cls** method to clear a form.
- Use the **PrintForm** method to send a bit-for-bit image of a form to the printer.

Lab Time



Go to the Getting Output portion of your lab manual.

Module 13: Data Access Using the Data Control

Σ Overview

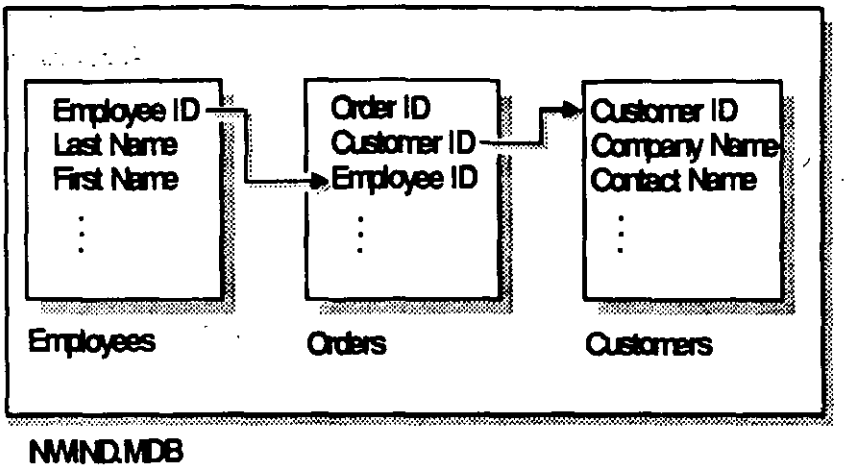
- Overview of a Database
- How Does Visual Basic Access Databases?
- The Data Control
- Binding Controls to the Data Control
- Data Control Walkthru
- Data Control Methods and Properties

Objectives

At the end of this module, you will be able to:

- Describe how Visual Basic accesses databases.
- Use the data control to view the contents of a Microsoft Access® database.

Overview of a Database



Overview of a Database

The following information will help you understand some of the terminology and concepts associated with database structure and design.

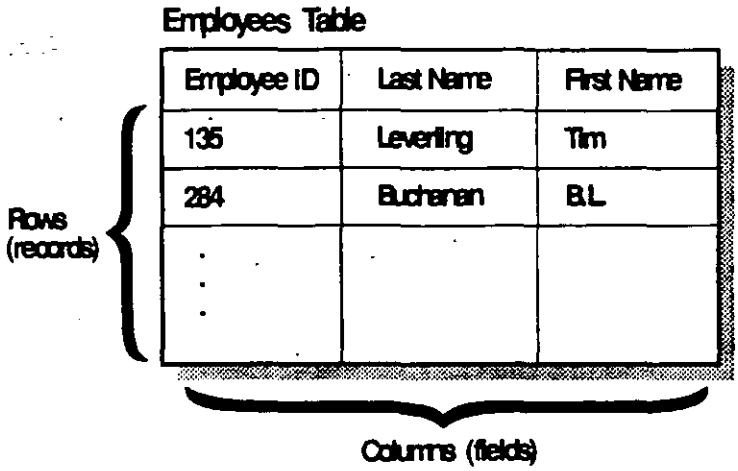
Relational Database Objects

Visual Basic provides a *relational* interface to database files. Basically, a relational database is one that stores data of *tables*, made up of *columns* and *rows*. In Visual Basic, columns are referred to as *fields* and rows are referred to as *records*.

Tables

A *table* is a logical grouping of related information. For example, the Northwind Traders database has a table that lists all the employees and another table that lists all the customers.

Overview of a Table



Fields

Each column or *field* in a table contains a single piece of information. For example, the Employees table has fields for Employee ID, Last Name, and so forth.

Records

A row or *record* in a table contains information about a single entry in a table. For example, a record in the Employees table would have information on a particular employee. Generally, you do not want two records in a table to have the exact same data. You would not want to have two Employees with the same name and the same ID number. Most tables have a field or combination of fields that must be unique.

Indexes

To make access to the database faster, most databases use *indexes*. Database table indexes are sorted lists that are faster to search than the tables.

Structure Query Language (SQL)

Once the data is stored in the database, retrieving it is made easier by using an English-like language called *Structured Query Language*, or *SQL*. SQL has evolved into the most widely accepted means to "converse" with a database. The user submits a query and the database returns all the rows that match that query.

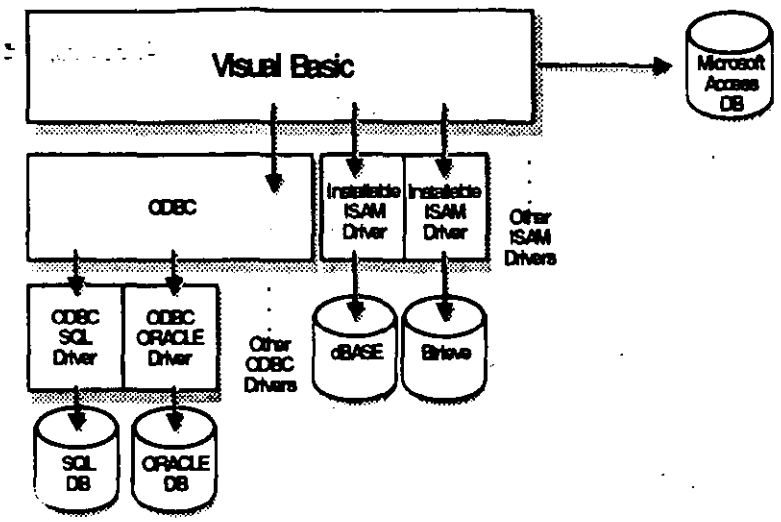
Example

```
Select [Last Name], Title From Employees Where Title = 'Sales Rep'
```

Sample Databases

In this course you will the Northwind Traders sample database. This database is included with Microsoft Access.

How Does Visual Basic Access Databases?



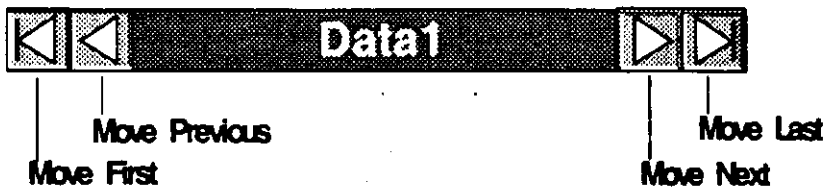
How Does Visual Basic Access Databases?

There are three types of databases that you can access from Visual Basic:

- "Native" Microsoft Access databases. These databases are accessed directly by Visual Basic.
- Indexed sequential access method (ISAM) databases—for example dBASE, Paradox®, and Btrieve® databases. Visual Basic reaches these databases through user-installable drivers that link Visual Basic to the specific databases.
- Open Database Connectivity (ODBC)–accessible databases. These include client-server database management systems (DBMSs), such as Microsoft SQL Server and ORACLE®. Visual Basic reaches these databases through the appropriate ODBC drivers.

There are various gateways that are available to connect to a mainframe database. This is typically implemented through an ODBC driver.

The Data Control



Data Control

The data control allows you to link a Visual Basic form to a database. With the data control, you can create an application that displays and updates data from a database—without writing a single line of code!

Adding the Data Control to the Toolbox

If the data control is not visible in the Toolbox, add the following line at the end of the [Visual Basic] section in the VB.INI file in the C:\WINDOWS directory:

```
DataAccess=1
```

The DatabaseName Property

The data control locates the database through the DatabaseName property. If you are connecting to a dBASE, Paradox, or Btrieve, database, set the DatabaseName property to the directory that contains the database files:

```
Data1.DatabaseName = "c:\walkthru\nwind.mdb"
```

The RecordSource Property

The RecordSource property indicates the name of a table, query or it contains the text of an SQL string.

The following example connects the data control to the Employees table:

```
Data1.RecordSource = "Employees"
```

The following example retrieves a subset of the Employees table.

```
Data1.RecordSource = "Select * from Employees where [Last Name] > 'M'"
```

Joining Two Tables

The following example joins two tables. The brackets are used with fields that have a space in the name.

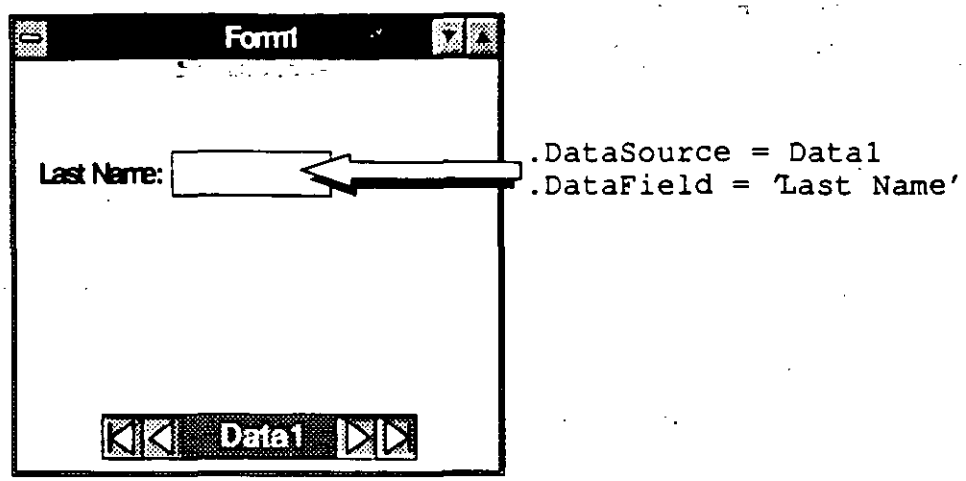
```
Data1.RecordSource =     "Select customers.[Customer Id],
                              ^[Contact Name], [Order Id]
                              ^From Customers, Orders
                              ^Where  Customers.[Customer Id] =
                              ^Orders.[Customer Id]"
```

Ordering the Records

The following example selects only some of the fields in the Employees table and orders the records by last name:

```
Data1.RecordSource = "Select [First Name], [Last Name] from
                          ^Employees Order by [Last Name]"
```

Binding Controls to the Data Control



Binding Controls to the Data Control

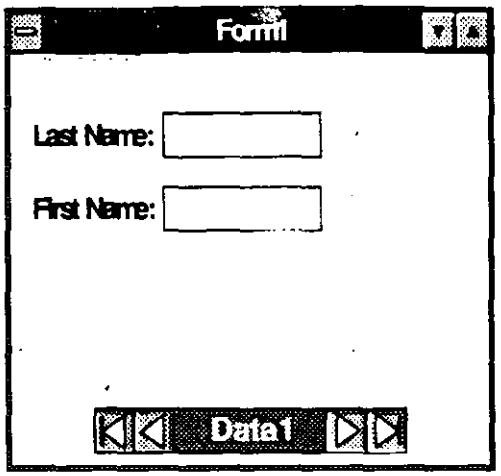
The text, picture, image, check box, masked edit, 3D panel, and 3D check box controls can be *bound* to a data control. When a control is bound to a data control, the data from the database is automatically displayed in the bound control. In addition, if the user changes the data in the bound control, those changes are automatically posted to the database as the user moves to another row.

To bind a control to a data control, set the DataSource property to the data control name and set the DataField property to a field name from the data control's table.

Example

```
'The DataSource property must be set at design time
Text1.DataSource = Data1

'The DataField property can be set at design or run time
Text1.DataField = "[Last Name]"
```

Data Control Walkthrough



Data Control Walkthrough

To get a quick overview of the data control, let's create a simple application that connects to the Microsoft Access Northwind database and browses the Employees table:

1. Start with a new project and add the data control to the form; leave the default name Data1.
2. Set the following properties for Data1:

   DatabaseName = c:\walkthru\nwind.mdb
   RecordSource = Employees

Note When you set the DatabaseName at design time, Visual Basic attempts to connect to the database. If it successfully connects to the database, it will display a list of possible values for the RecordSource property. Notice when you set the RecordSource property, you can type data in directly or choose the DOWN ARROW to display a list of possible selections.

---

3. Add two text boxes to the form; leave the names Text1 and Text2.
4. Set the following properties for Text1:

   DataSource = Data1
   DataField = Last Name

5. Set the following properties for Text2:

   DataSource = Data1
   DataField = First Name

6. Run your application!

(continued on following page)

7. Click the right arrow button to move forward, left arrow button to move backward.
8. Change a last name, click the right arrow button to move forward, and then click the left arrow button to move back. The record should be updated.
9. Notice you have not written a single line of code!

Data Control Methods and Properties

- Refresh Method
- Connect
- Exclusive
- ReadOnly

Refresh Method

The data control reflects modifications to existing data by other users but does not reflect records deleted or added by other users. The **Refresh** method updates the data control with the latest information from the database.

```
Data1.Refresh
```

Connect Property

The Connect property indicates the type of database that will be opened. The Connect property does not need to be set if you are connecting to a Microsoft Access database.

Database format	Database name	Connect
Microsoft Access	drive:\path\file.mdb	(none)
dBASE	drive:\path\	"dbase III;" or "dbase IV;"
Paradox	drive:\path\	"paradox;"
Btrieve	drive:\path\	"btrieve;"
ODBC	Registered data source name (server)	"odbc;dsn=*datasource*;uid=*user*;pwd=*password*" (Professional Edition only)

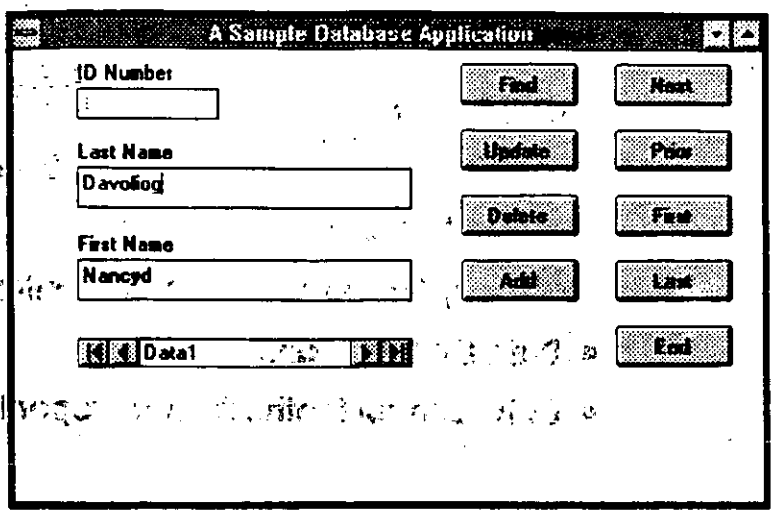
Exclusive Property

If you set the Exclusive property to **True** and then open the database, no other application will be allowed to open the database until you close the database. If another application has the database open and you attempt to open the database with Exclusive set to **True**, your application will receive a run-time error.

ReadOnly Property

If you set the ReadOnly property to **True**, your application will not be allowed to write to the database.

A Sample Application



A Sample Application

Here is a sample application that adds, finds, updates and deletes records from a database.

The details on how to code this application are beyond the scope of this class. However, you may find this application useful as an example.

The demonstration program DBSAMPLE.MAK is located in the \WALKTHRU\DBSAMPLE directory.

Σ Summary

- Overview of a Database
- How Does Visual Basic Access Databases?
- The Data Control
- Binding Controls to the Data Control
- Data Control Walkthru
- Data Control Methods and Properties