Microsoft Visual Basic Knowledge Base Articles
Prepared April 13, 1993

Microsoft Visual Basic Knowledge Base Articles

- **Frequently Asked Questions**
  - Keeping the Current Record the Same After Using Refresh
    - Copying the Current Database Record Into a Record Variable
    - Using a Data Control to Scroll Up and Down in a Recordset
    - DDE From Visual Basic to Excel For Windows
    - Example of Client/Server DDE Between Visual Basic Applications
    - How to Create Scrollable Viewports in Visual Basic
    - How to Determine When a Shelled Process has Terminated
    - How to Get Windows Master List (Task List) Using Visual Basic
    - How to Print a Form or Control using StretchDIBits
    - How to Use Windows BitBlt Function From Visual Basic
  - How VB Can Determine If a Specific Windows Program Is Running
    - LSTRCPY API Call to Receive LPSTR Returned From Other APIs
- **Data Access**
- **Dynamic Data Exchange (DDE)**
- **User Interface**
- **Controls (Text Boxes, Combo Boxes, Custom Controls)**
- **Printing**
- **Fonts**
- **Graphics**
- **CDK/DLLs (Control Development Kit/Dynamic Linked Libraries)**
- **Programming Issues**
- **Miscellaneous**

Visual Basic Knowledge Base Articles

Frequently Asked Questions

Data Access

- ODBC Setup & Connection Issues for Visual Basic Version 3.0
- Keeping the Current Record the Same After Using Refresh
- Copying the Current Database Record Into a Record Variable
- Using a Data Control to Scroll Up and Down in a Recordset

Dynamic Data Exchange (DDE)

User Interface

Controls (Text Boxes, Combo Boxes, Custom Controls)

Printing

Fonts

Graphics

CDK/DLLs (Control Development Kit/Dynamic Linked Libraries)

Programming Issues

Miscellaneous

Visual Basic Knowledge Base Articles

Frequently Asked Questions

Data Access

Dynamic Data Exchange (DDE)

- DDE Example Between Visual Basic and Windows Program Manager
- VB DDE to Excel with Embedded TAB Can Truncate String in Excel
- VB Example of Using DDE LinkExecute to Word for Windows 2.0
- Using DDE Between Visual Basic and Q+E for Windows
- DDE Example Between Visual Basic and Word for Windows
- How to Establish a Network DDE Link Using Visual Basic

User Interface

Controls (Text Boxes, Combo Boxes, Custom Controls)

Printing

Fonts

Graphics

CDK/DLLs (Control Development Kit/Dynamic Linked Libraries)

Programming Issues

Miscellaneous

Visual Basic Knowledge Base Articles

Frequently Asked Questions

Data Access

Dynamic Data Exchange (DDE)

User Interface

- How to Create a Form with no Title Bar in VB for Windows
  - How to Move Controls Between Forms in VB for Windows

Controls (Text Boxes, Combo Boxes, Custom Controls)

Printing

Fonts

Graphics

CDK/DLLs (Control Development Kit/Dynamic Linked Libraries)

Programming Issues

Miscellaneous

Visual Basic Knowledge Base Articles

Frequently Asked Questions

Data Access

Dynamic Data Exchange (DDE)

User Interface

Controls (Text Boxes, Combo Boxes, Custom Controls)

- How to Create Column and Row Labels in VB Grid Custom Control
- How to Read Flag Property of VB Common Dialog Custom Controls
- How to Use HORZ1.BMP with Professional Toolkit Gauge Control
- How to Use VB Graph Control to Graph Data from Grid Control
- PENCNTRL.VBX: "Requires Microsoft Windows for Pen Computing"
- VB AniButton Control: Cannot Resize if PictDrawMode=Autosize
- VB.EXE "License File for Custom Control Not Found" Explanation
- VB Graph Custom Control: DataReset Property Resets to 0 (Zero)
- VB Graph Control: ThisPoint, ThisSet Reset to 1 at Run Time
- VB Graph Control Displays Maximum of 80 Characters Per Title
- VB Key Status: Autosize Property Affects Height and Width
- VB MCI Control Does Not Support PC Speaker Driver
- VB MCI Control Does Not Support Recording of MIDI Data
- How to Close VB Combo Box with ENTER key
- How to Limit User Input in VB Combo Box with SendMessage API
- DEL Key Behavior Depends on Text Box MultiLine Property
- Determining Number of Lines in VB Text Box; SendMessage API
- Disabling the ENTER Key BEEP in a Visual Basic Text Box
- How to Scroll VB Text Box Programmatically and Specify Lines
- UCase$/LCase$ in Text Box Change Event Inverts Text Property

Printing

Fonts

Graphics

CDK/DLLs (Control Development Kit/Dynamic Linked Libraries)

Programming Issues

Miscellaneous

Visual Basic Knowledge Base Articles

Frequently Asked Questions

Data Access

Dynamic Data Exchange (DDE)

User Interface

Controls (Text Boxes, Combo Boxes, Custom Controls)

Printing

- VB Can Call Escape API to Specify Number of Copies to Printer
  - How to Set Landscape or Portrait for Printer in Windows 3.0
  - Using PASSTHROUGH Escape to Send Data Directly to Printer
  - Using an Escape to Obtain and Change Paper Size for Printer
  - How to Obtain & Change the Paper Bins for the Default Printer

Fonts

Graphics

CDK/DLLs (Control Development Kit/Dynamic Linked Libraries)

Programming Issues

Miscellaneous

Visual Basic Knowledge Base Articles

Frequently Asked Questions

Data Access

Dynamic Data Exchange (DDE)

User Interface

Controls (Text Boxes, Combo Boxes, Custom Controls)

Printing

Fonts

How to Use More than One Type of Font in Picture Box

VB Uses Bitmap Fonts when TrueType FontSize is Less than 8

Graphics

CDK/DLLs (Control Development Kit/Dynamic Linked Libraries)

Programming Issues

Miscellaneous

Visual Basic Knowledge Base Articles

Frequently Asked Questions

Data Access

Dynamic Data Exchange (DDE)

User Interface

Controls (Text Boxes, Combo Boxes, Custom Controls)

Printing

Fonts

Graphics

- Overflow Error Plotting Points Far Outside Bounds of Control
  - How to Make a Push Button with a Bitmap in Visual Basic
- How to Use FillPolygonRgn API to Fill Shape in Visual Basic
  - How to Send an HBITMAP to Windows API Function Calls from VB
  - How to Rotate a Bitmap in VB for Windows
  - How to Create a Transparent Bitmap Using Visual Basic
  - How to Copy Entire Screen into a Picture Box in VB for Windows
  - How to Flood Fill (Paint) in VB using ExtFloodFill Windows API

CDK/DLLs (Control Development Kit/Dynamic Linked Libraries)

Programming Issues

Miscellaneous

Visual Basic Knowledge Base Articles

Frequently Asked Questions

Data Access

Dynamic Data Exchange (DDE)

User Interface

Controls (Text Boxes, Combo Boxes, Custom Controls)

Printing

Fonts

Graphics

CDK/DLLs (Control Development Kit/Dynamic Linked Libraries)

- VB CDK: Example of Subclassing a Visual Basic Form
  - Declare Currency Type to Be Double When Returning from DLL
- How to Pass One-Byte Parameters from VB to DLL Routines
  - VB "Cannot Find DLL, Insert in Drive A" Using Shell
  - VB "Bad DLL Calling Convention" Means Stack Frame Mismatch
  - Diagnosing "Error in loading DLL" with LoadLibrary

Programming Issues

Miscellaneous

Visual Basic Knowledge Base Articles

Frequently Asked Questions

Data Access

Dynamic Data Exchange (DDE)

User Interface

Controls (Text Boxes, Combo Boxes, Custom Controls)

Printing

Fonts

Graphics

CDK/DLLs (Control Development Kit/Dynamic Linked Libraries)

Programming Issues

- How to Invoke Search in Windows Help from VB Program
  - How to Use Windows WNetAddConnection in Visual Basic
  - How to Invoke GetSystemMetrics Windows API Function from VB
  - How VB Can Get Windows Status Information via API Calls
  - How to Add a Horizontal Scroll Bar to Visual Basic List Box
  - How to Create a Flashing Title Bar on a Visual Basic Form
  - How to Set Windows System Colors Using API and Visual Basic
  - Creating TOPMOST or "Floating" Window in Visual Basic
  - Example of How to Read and Write Visual Basic Arrays to Disk
  - How to Determine Display State of a VB Form, Modal or Modeless
  - How to Determine the Number of VB Applications Running at Once
- How to Kill an Application with System Menu Using Visual Basic
  - How to Set Focus to First VB .EXE Instance When Second Invoked
  - How to Create a System-Modal Program/Window in Visual Basic
  - How to Access Windows Initialization Files Within Visual Basic
  - How to Create and Use a Custom Cursor in Visual Basic; Win SDK
  - How to Play a Waveform (.WAV) Sound File in Visual Basic
  - Visual Basic for Windows Reference Materials

Miscellaneous

Visual Basic Knowledge Base Articles

Frequently Asked Questions

Data Access

Dynamic Data Exchange (DDE)

User Interface

Controls (Text Boxes, Combo Boxes, Custom Controls)

Printing

Fonts

Graphics

CDK/DLLs (Control Development Kit/Dynamic Linked Libraries)

Programming Issues

Miscellaneous

- F5 in Run Mode with Focus on Main Menu Bar Acts as CTRL+BREAK
- Visual Basic SendKeys Statement Is Case Sensitive
- No New Timer Events During Visual Basic Timer Event Processing
- Scope of Line Labels/Numbers in Visual Basic for Windows
- Sending Keystrokes from Visual Basic to an MS-DOS Application
- Task List Switch to VB Application Fails After ALT+F4 Close
- Example of Sharing a Form Between Projects in VB for Windows
- "Property or Control Not Found" Using Form/Control Data Type
- Avoid Could not execute: SETUP1.EXE 2" Error, Use COMPRESS-r

Keeping the Current Record the Same After Using Refresh

Article ID: Q97181

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

Copying the Current Database Record Into a Record Variable

Article ID: Q97413

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

Using a Data Control to Scroll Up and Down in a Recordset

Article ID: Q97414

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

DDE From Visual Basic to Excel For Windows

Article ID: Q75089

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

Example of Client/Server DDE Between Visual Basic Applications

Article ID: Q74861

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

How to Create Scrollable Viewports in Visual Basic

Article ID: Q71068

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

How to Determine When a Shelled Process has Terminated

Article ID: Q96844

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

How to Get Windows Master List (Task List) Using Visual Basic

Article ID: Q78001

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

How to Print a Form or Control using StretchDIBits

Article ID: Q85978

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

How to Use Windows BitBlt Function From Visual Basic

Article ID: Q71104

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

How VB Can Determine If a Specific Windows Program Is Running

Article ID: Q72918

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

LSTRCPY API Call to Receive LPSTR Returned From Other APIs

Article ID: Q78304

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

ODBC Setup & Connection Issues for Visual Basic Version 3.0

Article ID: Q97415

Summary:

There are four possible problem areas that can contribute to a failure to connect to a database server when using ODBC and Visual Basic:

- Having correct .INI file settings.
- Having the correct DLLs in the right place.
- Having the server information needed to connect to a server correctly.
- Meeting the needs of Microsoft and Sybase SQL Servers.

More Information:

The following describes each of the four areas, giving possible errors and problems that can arise if things are not set up correctly.

INI file settings

There are two .INI files (ODBCINST.INI and ODBC.INI) that must reside in the Windows directory and must contain correct information about the installed ODBC drivers and servers.

ODBCINST.INI contains the ODBC driver information needed to register new servers using the RegisterDataBase() statement in Visual Basic. Here is an example .INI file for the SQL Server driver that ships with Visual Basic:

```
[ODBC Drivers]
SQL Server=Installed

[SQL Server]
Driver=C:\WINDOWS\SYSTEM\sqlsrvr.dll
Setup=C:\WINDOWS\SYSTEM\sqlsetup.dll
```

The [ODBC Drivers] section tells the driver manager the names of the installed drivers. The [SQL Server] section tells the ODBC driver manager the names of the dynamic link libraries (DLLs) to use to access data from a server set up as a SQL Server. The order of the two sections and their entries is arbitrary.

ODBC.INI contains the data for each installed driver. The driver manager uses this information to determine which DLL to use to access data from a particular database backend. Here is an example of a file containing three data sources all using the SQL Server driver:

```
[ODBC Data Sources]
MySQL=SQL Server
CorpSQL=SQL Server

[MySQL]
Driver=C:\WINDOWS\SYSTEM\sqlsrvr.dll
Description=SQL Server on server MySQL
OemToAnsi=No
Network=dbnmp3
Address=\\mysql\pipe\sql\query

[CorpSQL]
```

```
   Driver=C:\WINDOWS\SYSTEM\sqlsrvr.dll
   Description=SQL Server on server CorpSQL
   OemToAnsi=No
   Network=dbnmp3
   Address=\\corpsql\pipe\sql\query
```

The first section tells the driver manager which sections appearing below it define the data source. As you can see, each entry has a value (in this case, SQL Server) that matches a value from the ODBCINST.INI file.

If the information on a data source is incorrect or missing, you may get the following error:

```
   ODBC - SQLConnect failure 'IM002[Microsoft][ODBC DLL] Data source
          not found and no default driver specified'
```

If the DLL listed on the Driver=... line cannot be found or is corrupt, the following error may occur:

```
   ODBC - SQLConnect failure 'IM003[Microsoft][ODBC DLL] Driver
          specified by data source could not be loaded'
```

ODBC and Driver DLLs

The following DLLs must be on the path or in the Windows system directory in order for ODBC to be accessible from Visual Basic:

```
   ODBC.DLL     - driver manager
   ODBCINST.DLL - driver setup manager
   VBDB300.DLL  - Visual Basic programming layer
```

If VBDB300.DLL is missing or corrupt, you see the following error in Visual Basic when you try to run the application:

```
   ODBC Objects require VBDB300.DLL
```

If either the ODBC.DLL or ODBCINST.DLL file is missing or corrupt, you see the following error in Visual Basic when you try to run the application:

```
  Cannot Find ODBC.DLL, File not Found
```

The SQL Server driver requires the following files:

```
   SQLSRVR.DLL  - actual driver
   SQLSETUP.DLL - driver setup routines
   DBNMP3.DLL   - named pipe routines needed by SQL server
```

If the SQLSRVR.DLL is missing or corrupt, you see the following error when calling the OpenDataBase() function with a SQL Server data source:

```
   ODBC - SQLConnect failure 'IM003[Microsoft][ODBC DLL] Driver
          specified by data source could not be loaded'
```

If the SQLSETUP.DLL is missing or corrupt, you see the following error when calling the RegisterDataBase statement with SQL Server as the driver name:

```
   The configuration DLL (C:\WINDOWS\SYSTEM\SQLSETUP.DLL) for the ODBC
```

SQL server driver could not be loaded.

Server Information Needed to Connect to a Data Source

Certain information is needed to connect to a data source using the OpenDataBase() function. This information is obtainable from the server administrator in the case of SQL Server. The following is an example of a call to the OpenDataBase() function to connect to a SQL Server called CorpSQL as a user named Guest with password set to taco:

```
Dim db As DataBase
Set db = OpenDataBase( "corpsql", False, False, "UID=guest;PWD=taco")
```

If any of this information is missing, an ODBC dialog box appears to give a user a chance to supply the needed data. If the information is incorrect, the following error occurs:

```
ODBC - SQLConnect failure '28000[Microsoft][ODBC SQL Server Driver]
       [SQL Server]  Login failed'
```

Information Specific to Microsoft and Sybase SQL Servers

For Microsoft and Sybase SQL Servers, you need to add stored procedures to the server itself by running a batch file of SQL statements to make a Microsoft or Sybase SQL Server ODBC-aware. In other words, before you can run a Visual Basic ODBC application using the SQL Server driver, you must first update the ODBC catalog of stored procedures. These procedures are provided in the INSTCAT.SQL file. The system administrator for the SQL Server should install the procedures by using the SQL Server Interactive SQL (ISQL) utility.

If the INSTCAT.SQL file is not processed on the server, the following error occurs:

```
ODBC - SQL Connect Failure
"08001" [Microsoft ODBC SQL Server Driver]
'unable to connect to data source'number: 606'
```

To install the catalog stored procedures by using the INSTCAT.SQL file, run INSTCAT.SQL from the command line using ISQL. Do not use the SAF utility provided with SQL Server. Microsoft SAF for MS-DOS and OS/2 is limited to 511 lines of code in a SQL script, and INSTCAT.SQL contains more than 511 lines of code.

Run ISQL from the OS/2 command line using the following syntax. Enter the two lines as one, single line, and do not include the angle braces <>.

```
ISQL /U <sa login name > /n /P <password> /S <SQL server name> /i
      <drive: \path\INSTCAT.SQL > /o <drive:\path\output file name>
```

```
/U  The login name for the system administrator.
/n  Eliminates line numbering and prompting for user input.
/P  Password used for the system administrator. This switch is case
    sensitive.
/S  The name of the server to set up.
/i  Provides the drive and fully qualified path for the location of
    INSTCAT.SQL
```

/o Provides ISQL with an output file destination for results including error listings.

Here's an example (shown here on two lines but actually entered on one):

```
ISQL /U sa /n /P squeeze /S BLUEDWARF /i C: \SQL\INSTCAT.SQL /o
     C: \SQL OUTPUT.TXT
```

Keeping the Current Record the Same After Using Refresh

Article ID: Q97181

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

Copying the Current Database Record Into a Record Variable

Article ID: Q97413

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

Using a Data Control to Scroll Up and Down in a Recordset

Article ID: Q97414

Summary:

This article is in the main Visual Basic Help file (VB.HLP). To view it, choose Technical Support from the Help menu. Then choose Microsoft Knowledge Base. Choose this article from the list of articles.

DDE Example Between Visual Basic and Windows Program Manager

Article ID: Q76551

Summary:

This article demonstrates how to send dynamic data exchange (DDE) interface commands to the Microsoft Windows Program Manager from Microsoft Visual Basic for Windows using DDE.

The interface commands available through DDE with the Windows Program Manager are as follows:

```
CreateGroup(GroupName,GroupPath)
ShowGroup(GroupName,ShowCommand)
AddItem(CommandLine,Name,IconPath,IconIndex,XPos,YPos)
DeleteGroup(GroupName)
ExitProgman(bSaveState)
```

A full explanation of the above commands can be found in Chapter 22, pages 19-22 of the "Microsoft Windows Software Development Kit Guide to Programming" version 3.0 manual.

An application can also obtain a list of Windows groups from the Windows Program Manager by issuing a LinkRequest to the "PROGMAN" item.

More Information:

The following program demonstrates how to use four of the five Windows Program Manager DDE interface commands and the one DDE request:

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Create the following controls with the given properties on Form1:

Object	Name	Caption
TextBox	Text1	
Button	Command1	Make
Button	Command2	Delete
Button	Command3	Request

(In Visual Basic version 1.0 for Window set the CtlName Property for the above objects instead of the Name property.)

3. Add the following code to the Command1_Click event:

```
Sub Command1_Click ()
  Text1.LinkTopic = "ProgMan|Progman"
  Text1.LinkMode = 2              ' Establish manual link.

  Text1.LinkExecute "[CreateGroup(Test Group)]"
          ' Make a group in Windows Program Manager.

  Text1.LinkExecute "[AddItem(c:\vb\vb.exe, Visual Basic)]"
          ' Add an item to that group.
```

```
  Text1.LinkExecute "[ShowGroup(Test Group, 7)]"
          ' Iconize the group and focus to VB application.

  On Error Resume Next  ' Disconnecting link with Windows Program
  Text1.LinkMode = 0    ' Manager causes an error in Windows 3.0.
             ' This is a known problem with Windows Program Manager.
End Sub
```

4. Add the following code to the Command2_Click event:

```
Sub Command2_Click ()
  Text1.LinkTopic = "ProgMan|Progman"
  Text1.LinkMode = 2          ' Establish manual link.

  Text1.LinkExecute "[DeleteGroup(Test Group)]"
          ' Delete the group and all items within it.

  On Error Resume Next  ' Disconnecting link with Windows Program
  Text1.LinkMode = 0    ' Manager causes an error in Windows 3.0.
              ' This is a known problem with Windows Program Manager.
End Sub
```

5. Add the following code to the Command3_Click event:

```
Sub Command3_Click ()
  Text1.LinkTopic = "ProgMan|Progman"
  Text1.LinkItem = "PROGMAN"
  Text1.LinkMode = 2      ' Establish manual link.
  Text1.LinkRequest       ' Get a list of the groups.

  On Error Resume Next  ' Disconnecting link with Windows Program
  Text1.LinkMode = 0    ' Manager causes an error in Windows 3.0.
             ' This is a known problem with Windows Program Manager.
End Sub
```

5. Press the F5 key to run the program.

6. Choose the Make button, then choose the Delete button. Note the result.

7. Choose the Request button. This will put a list of the groups in the Windows Program Manager to be placed in the text box. The individual items are delimited by a carriage return plus linefeed.

As noted in the Windows Software Development Kit (SDK) manual mentioned above, the ExitProgman() command will only work if Windows Program Manager is NOT the shell (the startup program when you start Windows).

VB DDE to Excel with Embedded TAB Can Truncate String in Excel

Article ID: Q82157

Summary:

If you send strings containing TAB characters in a dynamic data exchange (DDE) conversation from Microsoft Visual Basic for Windows to Microsoft Excel, the string may be truncated in Excel if you specify a specific row and column in the Visual Basic for Windows LinkItem property. If you do not specify a column in the LinkItem property but only specify a specific row, your string will be parsed by Excel, and each TAB will cause the characters following the TAB to be entered into the following cell in Excel.

More Information:

This behavior occurs when the following is true:

- A string that you are trying to send to Excel through DDE contains an embedded TAB.

- You set your LinkItem property to a specific Excel cell (both row and column, such as R1C1, meaning row 1 column 1).

The attempted conversation will result in a truncated string. For example, if you pass the following string to Excel

```
"The cow jumped" + Chr$(9) + "over the moon"
```

and if the two conditions above are true, the only thing you will see on the Excel side is "The cow jumped". The rest of the string will be lost.

The following code example passes strings to Excel from a list box with TAB-delimited columns. Run the program twice, and uncomment the LinkItem line to see the different behavior.

Steps to Reproduce Behavior
---

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Put a text box on the form (Form1), and change the Name (change CtlName in Visual Basic version 1.0 for Windows) property to "ddebox".

3. Put a list box (List1) and a command button (Command1) on Form1.

4. Add the following code to the Form_Load procedure:

```
Sub Form_Load ()
  Form1.Show
  ' Add items to list box with TABs embedded.
  List1.AddItem "hey" + Chr$(9) + "is"
  List1.AddItem "for" + Chr$(9) + "horses"
End Sub
```

5. Add the following code to the Command1_Click event procedure:

```
Sub Command1_Click ()
    Const NONE = 0, COLD = 2     ' Define constants.

    If ddebox.LinkMode = NONE Then
        Z% = Shell("Excel", 4)   ' Start Excel.
       ' Set link topic.
        ddebox.LinkTopic = "Excel|Sheet1"
        ddebox.LinkItem = ""     ' Set link item.
        ddebox.LinkMode = COLD   ' Set link mode.
    End If

    ' Loop through all items in list box:
    For i% = 0 To List1.ListCount - 1
      Row$ = Format$(i% + 1)         ' Format row variable.
      ' ddebox.LinkItem = "R"+Row$   ' Take out comment to send entire
                                     ' string.
      ' Comment next line when uncommenting above line.
      ddebox.LinkItem = "R" + Row$ + "C1"   ' This statement truncates
                                            ' string in Excel.
      ddebox.text = List1.list(i%) ' Assign text box to list box string.
      ddebox.LinkPoke     ' Send the string to Excel.
    Next

    ddebox.LinkMode = NONE
End Sub
```

For best results, make sure Excel is not running before you start the program. When you start the program, notice the list box has the strings added to it during the form Load event. If you choose the command button to initialize the DDE conversation with the program typed in exactly as shown, the following will appear in Excel:

```
hey   ' This will be in cell A1.
for   ' This will be in cell A2.
```

If you change the assignment statement of the LinkItem of the ddebox from

```
   ddebox.LinkItem = "R" + Row$ + "C1"
```

to

```
   ddebox.LinkItem = "R"+ Row$
```

notice that the entire string is passed to Excel with the following results:

```
hey       is      ' These words will be in A1 and B1.
for        horses ' These words will be in A2 and B2.
```

The reason for this behavior is that Excel uses TABs as its delimiter. You can use this method to send multiple items to Excel, placing them in their own cells if desired. If that is not the desired result, you will have to make sure you compensate for the lost parts of the string.

VB Example of Using DDE LinkExecute to Word for Windows 2.0
Article ID: Q82879
Summary:

This article demonstrates how to send a LinkExecute event to Microsoft Word for Windows from Microsoft Visual Basic for Windows using dynamic data exchange (DDE).

The commands available through DDE with Word for Windows are as follows:

- Any Macro in Word for Windows
- Any embedded WordBasic command built into Word for Windows

A full explanation of the above commands can be found in Word for Windows online Help under the topic "WordBasic."

More Information:

The following example program demonstrates how to:

- Automatically start Word for Windows

- Automatically send text typed in a Visual Basic for Windows text box to the Word for Windows document

- Print the Word for Windows document to the selected printer.

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Create the following controls with the given properties on Form1:

Object	Name	Caption
TextBox	Text1	
Button	Command1	Start Word
Button	Command2	Link
Button	Command3	Send Text
Button	Command4	Print

(In Visual Basic version 1.0 for Windows set the CtlName Property for the above objects instead of the Name property.)

3. Add the following code to the Command1_Click event:

```
Sub Command1_Click ()
    x = Shell("winword.exe", 7) ' Start Word for Windows minimized
                                ' without the focus.
    x = DoEvents()          ' This gives WinWord time to load.
End Sub
```

4. Add the following code to the Command2_Click event procedure:

```
Sub Command2_Click ()
```

```
    text1.LinkMode = 0 ' Clears DDE link if it already exists.
    text1.LinkTopic = "WinWord|document1"   ' Sets up link with
                                            ' WINWORD.EXE.
    text1.LinkMode = 2        ' Establish a manual DDE link.
    text1.linktimeout = 60   ' Set the time for a response to 6 seconds.
    ' If a DDETIMEOUT occurs increase the Text1.Linktimeout.
    text1.LinkExecute "[InsertBookmark .Name ="+Chr$(34)+"Foo"+Chr$(34)+"]"
    '(Note that the space is necessary as shown before .Name in the above
    ' LinkExecute statement.)
    text1.LinkItem = "Foo"      ' Set link to a bookmark on document.
End Sub
```

5. Add the following code to the Command3_Click event procedure:

```
Sub Command3_Click ()
    text1.LinkPoke  ' Sends the contents of the text box.
End Sub
```

6. Add the following code to the Command4_Click event procedure:

```
Sub Command4_Click ()
    text1.LinkExecute "[FilePrintDefault]"   ' Prints the doc with the
                                             ' default printer settings.
End Sub
```

7. Press the F5 key to run the program.

6. Choose the Start Word button.

7. Choose the Link button. This will establish a DDE conversation with Word's Document1 and create a bookmark called Foo using LinkExecute and the embedded InsertBookmark WordBasic command. It will then set the LinkItem to this newly created bookmark in Document1.

8. Type some text in the text box and choose the Send Text command button to send the contents of the text box to Word for Windows.

9. Choose the Print button to print the document in Word for Windows.

Using DDE Between Visual Basic and Q+E for Windows

Article ID: Q75090

Summary:

This article describes how to initiate a Dynamic Data Exchange (DDE) conversation between a Microsoft Visual Basic for Windows destination application and a Pioneer Software Q+E for Windows source application. (Q+E is a database query tool.)

This article demonstrates how to:

1. Prepare a Q+E database file for active DDE.

2. Initiate a manual DDE link (information updated upon request from the destination) between Visual Basic for Windows (the destination) and Q+E (the source).

3. Use LinkRequest to update information in Visual Basic for Windows (the destination) based on information contained in Q+E (the source).

4. Initiate a automatic DDE link (information updated automatically from source to destination) between Visual Basic for Windows (the destination) and Q+E (the source).

5. Use LinkPoke to send information from Visual Basic for Windows (the destination) to Q+E (the source).

6. Change the LinkMode property between Automatic and Manual.

More Information:

A destination application sends commands through DDE to the source application to establish a link. Through DDE, the source provides data to the destination at the request of the destination or accepts information at the request of the destination.

The following steps serve as a example of how to establish a DDE conversation between Visual Basic for Windows and Q+E.

First, generate a Q+E database file to act as the source.

1. Create a database (.DBF) file (see the Q+E manuals for the procedure). For this example, you will use one of the default files, ADDR.DBF, that is provided with Microsoft Excel for Windows.

2. If Q+E is already running, exit Q+E. For this example to work properly, Q+E must not be loaded and running.

Next, create the destination application in Visual Basic for Windows.

The destination is the application that performs the link operations. It prompts the source to send information or informs the source that information is being sent.

1. Start Visual Basic for Windows. Form1 will be created by default.

2. Create the following controls with the following properties on

Form1:

Default Name	Caption	Name
Text1	(not applicable)	Text1
Option1	Manual Link	ManualLink
Option2	Automatic Link	AutomaticLink
Command1	Poke	Poke
Command2	Request	Request

(In Visual Basic version 1.0 for Windows, set the CtlName Property for the above objects instead of the Name property.)

3. Add the following code to the General Declaration section of Form1:

```
Const AUTOMATIC = 1
Const MANUAL = 2
Const NONE = 0

'  Const TRUE = -1  ' In Visual Basic 1.0 for Windows uncomment
'  Const FALSE = 0  ' these two lines.
```

4. Add the following code to the Load event procedure of Form1:

```
Sub Form_Load ()        ' This procedure will start Q+E and load the
                        ' file "ADDR.DBF".
    z% = Shell("C:\EXCEL\QE C:\EXCEL\QE\ADDR.DBF",1)
    z% = DoEvents ()                ' Process Windows events. This
                                    ' ensures that Q+E will be
                                    ' executed before any attempt is
                                    ' made to perform DDE with it.
    Text1.LinkMode = NONE           ' Clears DDE link if it already
                                    ' exists.
    Text1.LinkTopic = "QE|QUERY1"   ' Sets up link with Q+E.
    Text1.LinkItem = "R1C1"         ' Set link to first cell on
                                    ' spreadsheet.
    Text1.LinkMode = MANUAL         ' Establish a manual DDE link.
    ManualLink.Value = TRUE
End Sub
```

5. Add the following code to the Click event procedure of the Manual Link button:

```
Sub ManualLink_Click ()
        Request.Visible = TRUE       ' Make request button valid.
        Text1.LinkMode = NONE        ' Clear DDE Link.
        Text1.LinkMode = MANUAL      ' Reestablish new LinkMode.
End Sub
```

6. Add the following code to the Click event procedure of the AutomaticLink button:

```
Sub HotLink_Click ()
        Request.Visible = FALSE  ' No need for button with automatic link.
        Text1.LinkMode = NONE    ' Clear DDE Link.
        Text1.LinkMode = AUTOMATIC        ' Reestablish new LinkMode.
```

```
End Sub
```

7. Add the following code to the Click event procedure of the Request button:

```
Sub Request_Click ()
    ' With a manual DDE link this button will be visible and when
    ' selected it will request an update of information from the source
    ' application to the destination application.
    Text1.LinkRequest
End Sub
```

8. Add the following code to the Click event procedure of the Poke button:

```
Sub Poke_Click ()
    ' With any DDE link this button will be visible and when selected
    ' it will poke information from the destination application to the
    ' source application.
    Text1.LinkPoke
End Sub
```

You can now run the Visual Basic for Windows destination application from the Visual Basic for Windows environment (skip to step 4) or you can save the application and create an .EXE file and run that from Windows (continue to step 1):

1. From the File menu, save the Form and Project using the name CLIENT.

2. From the File menu, choose Make an EXE File, and name it CLIENT.EXE.

3. Exit Visual Basic for Windows.

4. Run the application (from Windows if an .EXE file, or from the Run menu if from the Visual Basic for Windows environment). Form1 of the destination application will be loaded and Q+E will automatically be started with the database file ADDR.DBF loaded.

5. Make sure that the main title bar in Q+E reads "Q + E," NOT "Q + E - ADDR.DBF." If the title bar is incorrect, then from the Window menu of Q+E, choose Arrange All.

You can now experiment with DDE between Visual Basic for Windows and Q+E for Windows:

1. Try typing some text in R1C1 (the cell that holds the name "Tyler") in the Q+E spreadsheet and then choose the Request button. The text will appear in the Visual Basic for Windows text box.

2. Choose the Automatic Link button and then type some more text in R1C1 of the Q+E spreadsheet. The text is automatically updated in the Visual Basic for Windows text box.

3. Type some text in the text box in the Visual Basic for Windows application and choose the Poke button. The text is sent to R1C1 in the Q+E spreadsheet.

Note that if you do not have the Allow Editing option checked on the Edit menu in Q+E, you will not be able to change the contents of the Q+E spreadsheet. This may prevent some DDE operations. For example, attempting to LinkPoke to Q+E from Visual Basic for Windows when the Allow Editing option is not chosen will cause the program to crash and result in a "Foreign application won't perform DDE method or operation" error message. Attempting to change the contents of the spreadsheet from Q+E will result in a "Use the allow editing command before making changes" error message. From the Edit menu of Q+E, choose Allow Editing to enable this option. When viewed from the Edit menu, Allow Editing should have a check mark next to it when enabled.

You can also establish DDE between applications at design time. For more information, see page 356 of the "Microsoft Visual Basic: Programmer's Guide," version 1.0 or Chapter 20 of the "Microsoft Visual Basic Programmer's Guide," version 2.0.

DDE Example Between Visual Basic and Word for Windows

Article ID: Q74862

Summary:

This article outlines the steps necessary to initiate dynamic data exchange (DDE) between a Microsoft Visual Basic application and a Microsoft Word for Windows (WINWORD.EXE) document at run time.

This article demonstrates how to:

- Prepare a Word for Windows document for active DDE.
- Initiate a manual DDE link (information updated upon request from the destination) between the Visual Basic application (the destination) and the document loaded into Word for Windows (the source).
- Use LinkRequest to update information in the Visual Basic destination based on information contained in the Word for Windows source.
- Initiate a automatic DDE link (information updated automatically from source to destination) between the Visual Basic destination and the Word for Windows source.
- Use LinkPoke to send information from the Visual Basic destination to the Word for Windows source.
- Change the LinkMode property between automatic and manual.

More Information:

A destination application sends commands through DDE to the source application to establish a link. Through DDE, the source provides data to the destination at the request of the destination or accepts information at the request of the destination.

Example Showing How to Establish a DDE Conversation

The steps below give an example of how to establish a DDE conversation between a Visual Basic application and a document loaded into Word for Windows (WINWORD.EXE).

Step One -- Create the Source Document in Word for Windows

1. Start Word for Windows. Document1 is created by default.

2. From the Window menu, choose Arrange All. This removes maximization if the document was maximized. Note that the title at the top of the WINWORD.EXE main title bar is now:

   Microsoft Word

   instead of:

   Microsoft Word - Document1

3. Press CTRL+SHIFT+END to select to the end of the document.

4. From the Insert menu, choose Bookmark. Under Bookmark Name, type:

   DDE_Link

   Press the ENTER key. This sets a bookmark for the entire document. This

bookmark functions as the LinkItem in the DDE conversation.

5. From the File menu, choose Save As, and save the document with the name SOURCE.DOC.

6. Exit from Word for Windows. For this particular example to function correctly, WINWORD.EXE must not be loaded and running.

Step Two -- Create the Destination Application in Visual Basic
---

1. Start Visual Basic. Form1 is created by default.

2. Create the following controls on Form1, giving the controls the properties shown in the table:

Default	NameCaption	Name
Text1	(Not applicable)	Text1
Option1	Manual Link	ManualLink
Option2	Automatic Link	AutomaticLink
Command1	Poke	Poke
Command2	Request	Request

3. Add the following code to the General Declaration section of Form1:

```
Const AUTOMATIC = 1
Const MANUAL = 2
Const NONE = 0
```

4. Add the following code to the Load event procedure of Form1:

```
Sub Form_Load ()
   'This procedure starts WINWORD.EXE, loads the document that was
   'created earlier, and prepares for DDE by creating a bookmark to
   'the whole document. This bookmark is necessary because it
   'functions as the LinkItem for the source in the DDE conversation.

   z% = Shell("WinWord Source.Doc",1)

   z% = DoEvents ()  'Process Windows events to ensure that
                     'WINWORD.EXE is executed before any attempt is
                     'made to perform DDE with it.

   Text1.LinkMode = NONE              'Clears DDE link if it exists.
   Text1.LinkTopic = "WinWord|Source" 'Sets up link with WINWORD.EXE.
   Text1.LinkItem = "DDE_Link"        'Set link to bookmark on document.
   Text1.LinkMode = MANUAL            'Establish a manual DDE link.
   ManualLink.Value = TRUE
End Sub
```

5. Add the following code to the Click event procedure of the Manual Link button:

```
Sub ManualLink_Click ()
   Request.Visible = TRUE  'Make request button valid.
   Text1.LinkMode = NONE   'Clear DDE Link.
   Text1.LinkMode = MANUAL 'Reestablish new LinkMode.
```

```
   End Sub
```

6. Add the following code to the Click event procedure of the Automatic Link button:

```
   Sub AutomaticLink_Click ()
      Request.Visible = FALSE     'No need for button with automatic link.
      Text1.LinkMode = NONE       'Clear DDE Link.
      Text1.LinkMode = AUTOMATIC  'Reestablish new LinkMode.
   End Sub
```

7. Add the following code to the Click event procedure of the Request button:

```
   Sub Request_Click ()
      'With a manual DDE link this button is visible. Clicking this button
      'requests an update of information from the source application to the
      'destination application.
      Text1.LinkRequest
   End Sub
```

8. Add the following code to the Click event procedure of the Poke button:

```
   Sub Poke_Click ()
      'With any DDE link, this button is visible. Clicking this button
      'pokes information from the destination application into the source
      'application.
      Text1.LinkPoke
   End Sub
```

Step Three -- Try it out

Now, you have two choices. You can run the Visual Basic destination application from the Visual Basic VB.EXE environment by skipping to step 4 below, or you can save the application, create an .EXE file, and run that from Windows by beginning with step 1 below.

1. From the File menu, choose Save, and save the form and project with the name DEST.

2. From the File menu, choose Make EXE File with the name DEST.EXE.

3. Exit from the Visual Basic environment (VB.EXE).

4. Run the application. Run an .EXE file from Windows, or if you're in the Visual Basic environment, from the Run menu, choose Start.

   Form1 of the Visual Basic destination application will be loaded, and Word for Windows will automatically start and load SOURCE.DOC.

5. Make sure that the main title bar in WINWORD.EXE reads "Microsoft Word," not "Microsoft Word - SOURCE.DOC." If the title bar is not correct, choose Arrange All from the Window menu.

Step Four -- Experiment

Experiment with DDE between Visual Basic and Word for Windows:

1. Try typing some text into the document in Word for Windows. Then click the Request button. The text appears in the text box.

2. Click the Automatic Link button. Then type some more text into the document in Word for Windows. The text is automatically updated in the Visual Basic text box.

3. Type some text in the text box in the Visual Basic application. Then click the Poke button. The text goes to the Word for Windows document.

Note that if in the WINWORD.EXE document, you delete the total contents of the bookmark, the bookmark is also deleted. Any attempt to perform DDE with this WINWORD.EXE session after deleting the bookmark causes this error:

   Foreign applications won't perform DDE method or operation.

If this happens, you must recreate the bookmark in the document in Word for Windows before performing any further DDE operations.

How to Establish a Network DDE Link Using Visual Basic

Article ID: Q93160

Summary:

This article demonstrates how to establish a network Dynamic Data Exchange (DDE) link between two computers running Microsoft Windows for Workgroups.

More Information:

Under DDE, a destination (or client) application sends commands through DDE to the source (or server) application to establish a link. Through DDE, the source provides data to the destination at the request of the destination or accepts information at the request of the destination. When you use DDE with Windows version 3.0 or 3.1 based applications, the source and destination applications are both located on the same computer.

When you use Network DDE with Windows for Workgroups based applications, DDE functions exactly the same way as standard DDE except that the source and destination applications are located on different computers. There are three steps involved in establishing a network DDE link.

Step One -- Add DDE Share by Calling NDdeShareAdd() Function

To establish a network DDE link, you must first establish a network DDE share for the conversation by calling the API NDdeShareAdd() function located in the NDDEAPI.DLL file. Here is the Visual Basic declaration:

```
Declare Function NDdeShareAdd Lib "NDDEAPI.DLL" (Server As Any, ByVal Level
   As Integer, ShareInfo As NDDESHAREINFO, ByVal nSize As Long) As Integer
```

Enter the entire statement as a single line. The first parameter is always a 0 and is passed with ByVal 0& from Visual Basic. The second parameter is always 2. The next parameter is a filled ShareInfo structure (given below). The last parameter is the size of the ShareInfo structure.

Here is The structure of the NDDESHAREINFO structure:

```
   Type NDDESHAREINFO
      szShareName As String * MAX_NDDESHARENAME_PLUSONE
      lpszTargetApp As Long    'LPSTR lpszTargetApp
      lpszTargetTopic As Long  'LPSTR lpszTargetTopic
      lpbPassword1 As Long     'LPBYTE lpbPassword1
      cbPassword1 As Long      'DWORD  cbPassword1;
      dwPermissions1 As Long   'DWORD  dwPermissions1;
      lpbPassword2 As Long     'LPBYTE lpbPassword2;
      cbPassword2 As Long      'DWORD  cbPassword2;
      dwPermissions2  As Long  'DWORD  dwPermissions2;
      lpszItem As Long         'LPSTR  lpszItem;
      cAddItems As Long        'LONG   cAddItems;
      lpNDdeShareItemInfo As Long
   End Type
```

The following table describes each field of the NDDESHAREINFO type:

Field Name	Purpose
szShareName	Name of the share to add.

lpszTargetApp	Pointer to null-terminated string containing the service or application name.
lpszTargetTopic	Pointer to null-terminated string holding the topic name
lpbPassword1	Pointer to the read-only password -- uppercase, null-terminated string. If null, pass null string, not zero.
cbPassword1	Length of read-only password
dwPermissions1	Full access password
cbPassword2	Length of the full access password
dwPermissions2	Permissions allowed by the full access password

Here are the permissions allowed for dwPermissions:

Name	Value	Function
NDDEACCESS_REQUEST	&H1	Allows LinkRequest
NDDEACCESS_ADVISE	&H2	Allows LinkAdvise
NDDEACCESS_POKE	&H4	Allows LinkPoke
NDDEACCESS_EXECUTE	&H8	Allows LinkExecute
NDDEACCESS_START_APP	&H10	Starts source application on connect

Here are the possible return values from NDdeShareAdd():

Name	Value	Meaning
NDDE_NO_ERROR	0	No error.
NDDE_BUF_TOO_SMALL	2	Buffer is too small to hold information.
NDDE_INVALID_APPNAME	13	Application name is not valid.
NDDE_INVALID_ITEMNAME	9	Item name is not valid.
NDDE_INVALID_LEVEL	7	Invalid level; nLevel parameter must be 2.
NDDE_INVALID_PASSWORD	8	Password is not valid.
NDDE_INVALID_SERVER	4	Computer name is not valid; lpszServer parameter must be NULL.
NDDE_INVALID_SHARE	5	Share name is not valid.
NDDE_INVALID_TOPIC	10	Topic name is not valid.
NDDE_OUT_OF_MEMORY	12	Not enough memory to complete request.
NDDE_SHARE_ALREADY_EXISTS	15	Existing shares cannot be replaced.

Step Two -- Create DDE source application

The following steps show you how to create a Visual Basic DDE source and destination application that communicates through a network DDE link.

1. From the DDE source computer, start Visual Basic or if Visual Basic is already running, from the File menu, choose New Project (ALT, F, N). Form1 is created by default.
2. Change the LinkTopic property of Form1 to VBTopic.
3. If you are running Visual Basic version 2.0 or 3.0 for Windows, change the LinkMode property of Form1 to 1 - Source. In Visual Basic version 1.0, this property is already set to 1 - Server; don't change it.
4. Add a text box (Text1) to Form1.
5. Change the Name property (CTlName in version 1.0) of Text1 to VBItem.
6. Add a timer (Timer1) to Form1.
7. From the File menu, choose New Module (ALT, F, M). Module1 is created.
8. Add the following code to the general declarations section of Module1, and enter all lines as a single line even though they may be shown on multiple lines for readability:

```
' DDE access options
Global Const NDDEACCESS_REQUEST = &H1
Global Const NDDEACCESS_ADVISE = &H2
Global Const NDDEACCESS_POKE = &H4
Global Const NDDEACCESS_EXECUTE = &H8
Global Const NDDEACCESS_START_APP = &H10
Global Const MAX_NDDESHARENAME_PLUSONE = 65
Type NDDESHAREINFO
   szShareName As String * MAX_NDDESHARENAME_PLUSONE
   lpszTargetApp As Long    'LPSTR lpszTargetApp
   lpszTargetTopic As Long  'LPSTR lpszTargetTopic
   lpbPassword1 As Long     'LPBYTE lpbPassword1
   cbPassword1 As Long      'DWORD  cbPassword1;
   dwPermissions1 As Long   'DWORD  dwPermissions1;
   lpbPassword2 As Long     'LPBYTE lpbPassword2;
   cbPassword2 As Long      'DWORD  cbPassword2;
   dwPermissions2  As Long  'DWORD  dwPermissions2;
   lpszItem As Long         'LPSTR  lpszItem;
   cAddItems As Long        'LONG   cAddItems;
   lpNDdeShareItemInfo As Long
End Type
Declare Function NDdeShareAdd Lib "NDDEAPI.DLL" (Server As Any, ByVal
   Level As Integer, ShareInfo As NDDESHAREINFO,
   ByVal Size As Long As Integer
Declare Function lstrcpy Lib "KERNEL" (szDest As Any, szSource As Any)
   As Long
'If using Visual Basic version 1.0, add the following constant declarations
'Global Const False = 0
'Global Const True = Not False
```

9. Add the following code to the Form_Load event of Form1:

```
Sub Form_Load ()
   Dim r As Integer
   Dim szShareName As String       ' Net DDE share name
   Dim szTargetName As String      ' Net DDE target name
   Dim szTopicName As String       ' Net DDE source topic name
   Dim szItemName As String
   Dim szReadOnlyPassword As String ' Read-only password Net DDE share
   Dim szFullAccessPassword As String ' Full access password
   Dim ShareInfo As NDDESHAREINFO
   Dim ShareInfoSize As Long
   Dim Result As Integer
   szShareName = "VBDDESource$" + Chr$(0)
   szTargetName = "VBTARGET" + Chr$(0)
   szTopicName = "VBTopic" + Chr$(0)
   szItemName = Chr$(0)                    'All items are allowed
   szReadOnlyPassword = Chr$(0)            'No password
   szFullAccessPassword = Chr$(0)
   'Provide the share, target, topic and item names along with passwords
   'that identify the network DDE share
   ShareInfo.szShareName = szShareName
   ShareInfo.lpszTargetApp = lstrcpy(ByVal szTargetName,
      ByVal szTargetName)
   ShareInfo.lpszTargetTopic = lstrcpy(ByVal szTopicName,
      ByVal szTopicName)
   ShareInfo.lpszItem = lstrcpy(ByVal szItemName, ByVal szItemName)
```

```
      ShareInfo.cbPassword1 = 0
      ShareInfo.lpbPassword1 = lstrcpy(ByVal szReadOnlyPassword,
         ByVal szReadOnlyPassword)
      ShareInfo.dwPermissions1 = NDDEACCESS_REQUEST Or NDDEACCESS_ADVISE Or
         NDDEACCESS_POKE Or NDDEACCESS_EXECUTE Or NDDEACCESS_START_APP
      ShareInfo.cbPassword2 = 0
      ShareInfo.lpbPassword2 = lstrcpy(ByVal szFullAccessPassword,
         ByVal szFullAccessPassword)
      ShareInfo.dwPermissions2 = NDDEACCESS_REQUEST Or NDDEACCESS_ADVISE Or
         NDDEACCESS_POKE Or NDDEACCESS_EXECUTE Or NDDEACCESS_START_APP
      ShareInfo.lpNDdeShareItemInfo = 15
      Result = NDdeShareAdd(ByVal 0&, 2, ShareInfo, Len(ShareInfo))
      ' Start the timer that will continually update the text box and
      ' the DDE link item with random data.
      timer1.Interval = 1000
      timer1.Enabled = True
   End Sub
```

10. Add the following code to the Timer1_Timer event procedure:

```
    Sub Timer1_Timer ()
       ' Display random value 0 - 99 in the text box (DDE source data).
       Randomize Timer
       VBItem.Text = Format$(Rnd * 100, "0")
    End Sub
```

11. From the File menu, choose Make EXE File...
12. Name the file VBTARGET.EXE and choose OK to create the .EXE file.
13. From the File Manager or Program Manager, run VBTARGET.EXE to display a random value in the text box every second.

Step Three -- Create the DDE destination application
---

14. From the DDE destination computer, start Visual Basic or if Visual Basic is already running, from the File menu, choose New Project (ALT, F, N). Form1 is created by default.
15. Add a text box (Text1) to Form1.
16. Add the following code to the Form_Load event of Form1:

```
    Sub Form_Load ()
       Dim r As Long
       Dim szComputer As String     ' Network server name.
       Dim szTopic As String
       ' Identify the network server where the DDE source application
       ' is running. The following statement assumes the source computer
       ' name is COMPUTER1. Change it to your source computer name.
       szComputer = "\\COMPUTER1"
       ' Identify the DDE share established by the source application
       szTopic = "VBDDESource$"
       Text1.LinkMode = 0
       ' The link topic identifies the computer name and link topic
       ' as established by the DDE source application
       Text1.LinkTopic = szComputer + "\" + "NDDE$" + "|" + szTopic
       Text1.LinkItem = "VBItem" ' Name of text box in DDE source app
       Text1.LinkMode = 1        ' Automatic link.
    End Sub
```

```
    'For this program to work, set the szComputer variable (above) to the
    'computer name that holds the DDE source application. Find the computer
    'name in the Network section of Windows for Workgroups Control Panel.
```

17. From the Run menu, choose Start to run the program.

You should see the same random values generated on the source computer displayed in the text box of the destination computer. If you receive the error message "DDE method invoked with no channel open" on the Text1.LinkMode = 1 statement in Step 16, make sure the szComputer variable is set correctly.

How to Create a Form with no Title Bar in VB for Windows

Article ID: Q83349

Summary:

To create a Microsoft Visual Basic for Windows form with a border but with no title bar, the Caption property of a form must be set to a zero-length string; the BorderStyle property must be set to Fixed Single (1), Sizable (2) or Fixed Double; and the ControlBox, MaxButton and MinButton properties must be set to False (0). If any text (including spaces) exists for the Caption property or if the ControlBox, MaxButton, or MinButton property is set to True, a title bar will appear on the form. Note that setting the BorderStyle property to None (0) will always result in a form with no title bar.

More Information:

Even with the ControlBox, MaxButton, and MinButton properties of a form set to False (0) and the BorderStyle set to Fixed Single (1), Sizable (2) or Fixed Double (3), the form will still have a title bar unless the Caption property is set to null. Setting the Caption to blanks will leave a title bar with no title.

Steps to Reproduce Behavior
---

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. From the Properties bar, set the ControlBox, MaxButton, and MinButton properties to False.

3. Set the Caption property to at least one space.

4. Press the F5 key to run the program. The form will have a title bar without a title.

5. Press CTRL+BREAK to return to design mode.

6. Set the Caption property to a zero-length string (that is, delete all characters including spaces).

7. Press the F5 key to run the program. There should be no title bar on the form.

You can also have a form with no title bar by setting the BorderStyle property to None (0).

How to Move Controls Between Forms in VB for Windows

Article ID: Q79884

Summary:

Microsoft Visual Basic for Windows does not support the actual movement of controls between forms. Attempting to change the parent/child relationship of a control from one form to another can result in unpredictable behavior.

However, by creating a control array of the same control type on each form, and by creating a subroutine or function in a Visual Basic for Windows module, you can simulate the movement of a control from one form to another. An example of how to do this is listed below.

More Information:

This example uses the Windows API functions GetFocus and GetParent to determine the origin of the control dropped onto a form.

The following steps demonstrate how to simulate the movement of controls between two forms. Note that you can improve this example by Loading and Unloading the controls as they are needed.

1. Start Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 will be created by default.

2. From the File menu, choose New Form (press ALT, F, F). Form2 will be created.

3. From the File menu, choose New Module (press ALT, F, M). Module1 will be created.

4. Create the following controls for both Form1 and Form2:

Control	Name	Property Setting
Command button	Command1()	Index = 0
Command button	Command2	Caption = "Enable Drag"

   (In Visual Basic version 1.0 for Windows, set the CtlName Property for the above objects instead of the Name property.)

5. Add the following code to the Module1 (or GLOBAL.BAS in Visual Basic version 1.0 for Windows):

```
' Windows API function declarations.
Declare Function GetFocus Lib "USER" () As Integer
Declare Function GetParent Lib "USER" (ByVal hWnd As Integer) As Integer
```

6. Add the following code to the General Declarations section of Form1:

```
    Dim EnableDrag As Integer
```

7. Add the following code to the Form_Load event procedure of Form1:

```
Sub Form_Load ()

   ' Move the form to the left half of the screen.
   Move 0, Top, Screen.Width \ 2
   Form2.Show
   EnableDrag = 0
   Command1(0).Top = 0
   Command1(0).Left = 100

   For i% = 1 To 4                              ' Load Control Array.
      Load Command1(i%)
      Command1(i%).Left = Command1(i% - 1).Left
      Command1(i%).Top = Command1(i% - 1).Top + Command1(i% - 1).Height
   Next i%

   For i% = 0 To 4                           ' Define Control Properties.
      Command1(i%).Caption = "Button" + Str$(i%)
      Command1(i%).Visible = -1
   Next i%
End Sub
```

8. Add the following code to the Command1_Click event procedure of Form1:

```
Sub Command1_Click (Index As Integer)
   Button_Clicked Command1(Index)        ' Call Routine in MODULE1.BAS.
End Sub
```

9. Add the following code to the Command2_Click event procedure of Form1:

```
Sub Command2_Click ()
   If EnableDrag = 0 Then                       ' Toggle DragMode.
      EnableDrag = 1
      Command2.Caption = "Disable Drag"
   Else
      EnableDrag = 0
      Command2.Caption = "Enable Drag"
   End If

   For i% = 0 To 4                           ' Set DragMode for Controls.
      Command1(i%).DragMode = EnableDrag
   Next i%
End Sub
```

10. Add the following code to the Form_DragDrop event procedure of Form1:

```
Sub Form_DragDrop (Source As Control, X As Single, Y As Single)
   Source.SetFocus                       ' Get Parent of Source Control.
   CtrlHnd% = GetFocus()
   Parent% = GetParent(CtrlHnd%)

   If Parent% <> Form1.hWnd Then       ' If Parent is other Form.
      Index% = Source.Index
      Command1(Index%).Caption = Source.Caption
```

```
      Command1(Index%).Left = Source.Left
      Command1(Index%).Top = Source.Top
      Command1(Index%).Width = Source.Width
      Command1(Index%).Height = Source.Height
      Command1(Index%).Visible = -1
      Source.Visible = 0
   End If
End Sub
```

11. Add the following code to the General Declarations section of Form2:

```
Dim EnableDrag As Integer
```

12. Add the following code to the Form_Load event procedure of Form2:

```
Sub Form_Load ()
   ' Move the form to the right half of the screen.
   Move Screen.Width \ 2, Top, Screen.Width \ 2

   EnableDrag = 0
   Command1(0).Visible = 0
   For i% = 1 To 4                        ' Load Control Array.
      Load Command1(i%)
      Command1(i%).Top = Command1(i% - 1).Top + Command1(i% - 1).Height
      Command1(i%).Visible = 0
   Next i%
End Sub
```

13. Add the following code to the Command1_Click event procedure of Form2:

```
Sub Command1_Click (Index As Integer)
   Button_Clicked Command1(Index)
End Sub
```

14. Add the following code to the Command2_Click event procedure of Form2:

```
Sub Command2_Click ()
   If EnableDrag = 0 Then
      EnableDrag = 1
      Command2.Caption = "Disable Drag"
   Else
      EnableDrag = 0
      Command2.Caption = "Enable Drag"
   End If

   For i% = 0 To 4
      Command1(i%).DragMode = EnableDrag
   Next i%
End Sub
```

15. Add the following code to the Form_DragDrop event procedure of Form2:

```
Sub Form_DragDrop (Source As Control, X As Single, Y As Single)
```

```
   Source.SetFocus           ' Determine Parent of Source.
   CtrlHnd% = GetFocus()
   Parent% = GetParent(CtrlHnd%)
   If Parent% <> Form2.hWnd Then
      Index% = Source.Index
      Command1(Index%).Caption = Source.Caption
      Command1(Index%).Left = Source.Left
      Command1(Index%).Top = Source.Top
      Command1(Index%).Width = Source.Width
      Command1(Index%).Height = Source.Height
      Command1(Index%).Visible = -1
      Source.Visible = 0
   End If
End Sub
```

16. Add the following code to Module1:

```
Sub Button_Clicked (Source As Control) ' Generic Click routine.
    MsgBox "Button" + Str$(Source.Index) + " Clicked!!!"
End Sub
```

17. From the Run menu, choose Start (press ALT, R, S) to run the program.

To drag controls from one form to the other, choose the Enable Drag button. Once this button has been activated on a form, you can drag any of the command buttons from one form to the other. The drag mode can be disabled by choosing the Disable Drag button. When drag mode has been disabled, clicking on any of the command buttons on the form will cause a message box to be displayed.

How to Create Column and Row Labels in VB Grid Custom Control

Article ID: Q84113

Summary:

The example program below demonstrates how you can display labels in the top row and left column of the Grid custom control at run time. It is not possible to assign labels in a grid at design time.

More Information:

The example program below assigns labels to a grid from the Form_Load event procedure. It puts numbers down the left, labeling the first non-fixed row as "1". It puts letters across the top, labeling the first 26 non-fixed columns as "A" through "Z" then subsequent columns with "AA", "AB", and so on.

Steps to Create Example Program
---

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. From the File menu, choose Add File. In the Files box, select the GRID.VBX. The Grid tool appears in the Toolbox.

3. Select the Grid tool from the Toolbox, and place a grid (Grid1) on Form1.

4. On the Properties bar, set the Grid Cols and Rows properties to 30.

5. Double-click the form to open the Code window. In the Procedure box, select Load. Enter the following code:

```
Sub Form_Load ()
    Dim i As Integer

    ' Make sure grid has at least one fixed column and row.
    If Grid1.FixedCols < 1 Or Grid1.FixedRows < 1 Then
        Stop
    End If

    ' Put letters across top.
    For i = 0 To Grid1.Cols - 2
        Grid1.Col = i + 1
        Grid1.Row = 0
        Grid1.Text = Chr$(i Mod 26 + Asc("A"))
        ' If more than 26 columns, use double letter labels.
        If i + Asc("A") > Asc("Z") Then
            Grid1.Text = Chr$(i \ 26 - 1 + Asc("A")) + Grid1.text
        End If
        Grid1.FixedAlignment(Grid1.Col) = 2  ' Centered.
    Next

    ' Put numbers down left edge.
    For i = 1 To Grid1.Rows - 1
        Grid1.Col = 0
```

```
        Grid1.Row = i
        Grid1.Text = Format$(i)
    Next
    Grid1.FixedAlignment(0) = 2  ' Centered.
End Sub
```

6. Press the F5 key to run the program.

How to Read Flag Property of VB Common Dialog Custom Controls

Article ID: Q84068

Summary:

The Flags property of a Common Dialog control can be read by examining individual bit values of the Flag property and comparing them with the predefined constant values in CONSTANT.TXT (or CONST2.TXT for Visual Basic version 1.0 for Windows). This applies to the following Visual Basic for Windows Common Dialogs:

- File Open Dialog
- File Save Dialog
- Color Dialog
- Choose Font Dialog
- Print Dialog

More Information:

The Flags property can be set at design time or run time.

To set the value of the Flags property, assign it a value. This is most commonly done using a predefined constant (found in CONSTANT.TXT or CONST2.TXT). For example, to set the PRINTTOFILE flag on the Print Dialog box, use the following code:

```
CMDialog1.Flags = PD_PRINTTOFILE
```

To set more than one flag, OR the two flags (the pipe [|] character acts the same as the OR statement):

```
CMDialog1.Flags = PD_PRINTTOFILE | PD_SHOWHELP
```

The settings of the Flags property can also be changed at run time by the user making various selections in the dialog box. When a selection is made, or the status of a check box or option button is changed, the Flags property reflects this change. You can then read the value of the Flags property and determine if a specific flag has been set.

For example, in the above sample code, two flags are set in the Flags property. The value of PD_PRINTTOFILE = &H00000020& and the value of PD_SHOWHELP = &H00000800&.

The binary equivalent of the two is the following:

```
PD_PRINTTOFILE = 00000000000000000000000000100000
PD_SHOWHELP    = 00000000000000000000100000000000
```

Thus the value:

```
Flags          = 00000000000000000000100000100000
```

Note how each flag setting has its own bit setting within the Flags property.

To determine if a specific flag is set, you only need to AND the flag with the Flags property. If the result is 0, then the flag is not set; if the result is the same as the flag value, then the flag is set.

For example:

```
   Form1.Print (CMDialog1.Flags AND PD_PRINTTOFILE)
```

The output is decimal 32. Thus, broken down:

```
   Flags          = 00000000000000000000100000100000
                             AND
   PD_PRINTTOFILE = 00000000000000000000000000100000
                    ----------------------------------
   Result         = 00000000000000000000000000100000
```

Thus, the flag for PRINTTOFILE is one of the flags that are set in the Flags property:

```
If  (CMDialog1.Flags AND PD_PRINTTOFILE) Then
   ' Code for printing to file goes here.
Else
   ' Code for printing to printer goes here.
End If
```

How to Use HORZ1.BMP with Professional Toolkit Gauge Control
Article ID: Q81459
Summary:

This article contains a program example of using the Visual Basic for Windows Gauge custom control (GAUGE.VBX) with the HORZ1.BMP bitmap file.

More Information:

Note: The GAUGE.VBX custom control file can be found in the \Windows\System subdirectory. The HORZ1.BMP bitmap file can be found in the \BITMAPS\GUAGE subdirectory that was created during installation.

Example Program
---

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. From the File menu, choose Add File. In the Files box, select the GAUGE.VBX custom control file. The Gauge tool will appear in the toolbox.

3. Create the following controls for Form1:

Control	Name	Property Setting
Timer	Timer1	Interval = 1
Gauge	Gauge1	Picture = "Horz1.BMP"
		Max = 50
		InnerBottom = 16
		InnerLeft = 38
		InnerRight = 2
		InnerTop = 14
		ForeColor = &HFF&

   (In Visual Basic version 1.0 for Windows, set the CtlName Property for the above objects instead of the Name property.)

4. Add the following line to the General Declarations section:

```
Dim YoYo As Integer
```

5. Add the following code to the Form_Load event procedure:

```
Sub Form_Load ()
   Form1.Caption = "YoYo Gauge Demo"
   Gauge1.Value = Gauge1.Min
End Sub
```

6. Add the following code to the Timer1_Timer event procedure:

```
Sub Timer1_Timer ()
   If Gauge1.Value = Gauge1.Max Then YoYo = -1
   If Gauge1.Value = Gauge1.Min Then YoYo = 1
```

```
   Gauge1.Value = Gauge1.Value + YoYo
End Sub
```

When run, this program example will alternately fill and empty the gauge control's fill area, as controlled by the Timer event procedure.

How to Use VB Graph Control to Graph Data from Grid Control

Article ID: Q84063

Summary:

This article contains an example of how to use a Graph custom control to graph the data contained in a Grid custom control.

In order to use either the Grid or the Graph control, you must add them to the Toolbox in the Visual Basic for Windows environment (in VB.EXE). You do this by selecting Add File from the File menu. From here select the Graph.VBX file, and then repeat the process for Grid.VBX. Graph.VBX and Grid.VBX should be found in your Windows\System directory.

More Information:

To create the example, do the following:

1. Run Visual Basic for Windows, or from the File menu, choose New Project (ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. From the File menu, choose Add File. In the Files box, select the GRAPH.VBX custom control file. The Graph tool appears in the Toolbox.

3. Repeat step 2 for the GRID.VBX custom control file.

4. Add a Grid control (Grid1), a Graph control (Graph1), and a command button (Command1) to Form1.

5. In the Load event for Form1, add the following code:

```
Sub Form_Load ()
   ' This Sub will do all the configuration for the Grid.
    ConfigureGrid
   ' This Sub will do all the configuration for the Graph.
    ConfigureGraph
End Sub
```

6. Create the following subroutine in the general Declarations section of Form1 to make it callable from anywhere in the form:

```
 Sub ConfigureGrid ()

     ' Set the number of cols and rows for the grid.
     Grid1.Rows = 11
     Grid1.Cols = 4

     ' Set the alignment for the fixed col to centered.
     Grid1.FixedAlignment(0) = 2

     ' Set the alignment for the variable cols to centered.
     Grid1.ColAlignment(1) = 2
     Grid1.ColAlignment(2) = 2
     Grid1.ColAlignment(3) = 2

     Grid1.ScrollBars = 0
```

```
        ' Add the row labels.
        Grid1.Col = 0
        For i = 1 To 10
            Grid1.Row = i
            Grid1.Text = Str$(i)
        Next i

         ' Add the Col labels.
        Grid1.Row = 0
            Grid1.Col = 1
            Grid1.Text = "May"
            Grid1.Col = 2
            Grid1.Text = "June"
            Grid1.Col = 3
            Grid1.Text = "July"

      ' Set the starting cell on the Grid.
        Grid1.Row = 1
        Grid1.Col = 1
    End Sub
```

7. Create the following subroutine in the general Declarations section of Form1 to make it callable from anywhere on the form:

```
    Sub ConfigureGraph ()
      ' Set the Graph to auto increment.
        Graph1.AutoInc = 1
        Graph1.BottomTitle = "Months"
        Graph1.GraphCaption = "Graph Caption"
      ' Set the number of data groupings.
        Graph1.NumPoints = 10
      ' Set the number of data points per group.
        Graph1.NumSets = 3
    End Sub
```

8. Place the following line of code into the KeyPress event for Grid1:

```
    Sub Grid1_KeyPress (KeyAscii As Integer)
      ' This adds each keystroke to the data in the current cell.
        Grid1.Text = Grid1.Text + Chr$(KeyAscii)
    End Sub
```

9. For the Click event of Command1, enter the following code:

```
    Sub Command1_Click ()
   ' This Sub graphs the data in the Grid using the Graph control.
      ' Set the graph to the first point.
        Graph1.ThisSet = 1
        Graph1.ThisPoint = 1
    ' Load the GraphData array with all the values from the Grid,
    '    in order.
        For i = 1 To 3
            For j = 1 To 10
                Grid1.Row = j
                Grid1.Col = i
                Graph1.GraphData = Val(Grid1.Text)
```

```
            Next j
        Next i

      ' This actually graphs the array to the Graph control.
        Graph1.DrawMode = 2
    End Sub
```

This example will give you a grid with three columns (Months) and 10 rows. After you enter the data into the columns, choose the command button (with the mouse or keys). The data will be taken from the grid and graphed as a line graph.

PENCNTRL.VBX: "Requires Microsoft Windows for Pen Computing"

Article ID: Q83800

Summary:

The Microsoft Professional Edition of Microsoft Visual Basic versions 2.0 and 3.0 for Windows, and Microsoft Professional Toolkit for Microsoft Visual Basic programming system version 1.0 for Windows, includes a custom control that gives you easy access to writing applications for Microsoft Windows for Pen Computing. Without Microsoft Windows for Pen Computing, PENCNTRL.VBX cannot be loaded into the Visual Basic for Windows programming environment.

If you try to load the PENCNTRL.VBX custom control without having installed Microsoft Windows for Pen Computing, the process will abort with the following message box:

   This program requires Microsoft Windows for Pen Computing

More Information:

For more information about Microsoft Windows for Pen Computing, call Microsoft End User Sales and Service at (800) 426-9400. If calling from outside the United States, contact your local Microsoft subsidiary.

VB AniButton Control: Cannot Resize if PictDrawMode=Autosize

Article ID: Q82159

Summary:

Resizing an Animated Button custom control by setting the Width or Height property at run time will not work if the PictDrawMode property is set to Autosize (1). This is by design. When the PictDrawMode property is in autosize mode, the size is determined by the size of the images loaded, not by the design time setting of Width or Height nor the run time setting of those values.

More Information:

Steps to Reproduce Behavior
---

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. From the Files menu, choose Add File. In the Files box, select the ANIBUTON.VBX custom control file. The Animated Button tool appears in the toolbox.

3. Add the following code to the Form_Load procedure:

```
Sub Form_Load ()
  Form1.BackColor = &HFFFF00  ' To make the size of the control more
                              ' visible.
  AniButton1.Move Form1.Width \ 4, 0, 1600, 1600
  AniButton1.TextPosition = 3  ' Put caption at top for clarity.
End Sub
```

4. Add the following code to the Form_Click procedure:

```
   Sub Form_Click ()
       AniButton1.Caption = "This is a very very long caption"
       AniButton1.PictDrawMode = 1   ' Autosize control.
      'AniButton1.PictDrawMode = 0   ' As Defined.
      'AniButton1.PictDrawMode = 2   ' Stretches image to fit.
   End Sub
```

4. Add the following code to the Form_DoubleClick event:

```
   Sub Form_DblClick ()
      Print AniButton1.Width
      AniButton1.Width = 400
      Print AniButton1.Width
      Print AniButton1.PictDrawMode
   End Sub
```

5. Run the project with the PictDrawMode setting of 0 uncommented and the other two commented out.

6. Click once to see the effect of changing the mode. Then double-click the form to see the changes due to changing the Width property. Because the caption is the largest object in an unloaded

Animated Button, the autosize adjusts to it.

7. Access the Frame property and load a bitmap into the first frame and an icon in the second, or vice versa.

8. Repeat steps 5 and 6. Notice that the larger object (the bitmap) causes the control to resize to it.

VB.EXE "License File for Custom Control Not Found" Explanation

Article ID: Q81458

Summary:

If you distribute the source code (.FRM) of a program that uses a custom control, you must also distribute the necessary custom control files for that control (.VBX, .DLL, and/or .EXE support files).

If a user has not purchased the Professional Edition of Microsoft Visual Basic versions 2.0 or 3.0 for Windows, or the Microsoft Professional Toolkit for Microsoft Visual Basic programming system version 1.0 for Windows, and the user receives a program containing an .FRM file written with the Professional Edition or Professional Toolkit, then the Visual Basic for Windows programming environment (VB.EXE) will not be able to load the program, and will display the following error message:

> License file for custom control not found. You do not have an appropriate license to use this custom control in the design environment.

Note that anyone who acquires a program in the form of an executable (.EXE) file that uses the custom controls from versions 2.0 or 3.0 of the Professional Edition of Visual Basic for Windows, or from version 1.0 of the Professional Toolkit for Visual Basic for Windows, will be able to run that program with no error.

More Information:

The licensing file, VB.LIC is installed by the SETUP.EXE program included in the Professional Edition of Visual Basic for Windows, or the SETUP.EXE included in the Visual Basic for Windows Professional Toolkit. This licensing file is installed into the Windows' \SYSTEM subdirectory. You are NOT allowed to distribute this file with any application that you develop and distribute.

A custom control's startup code checks to see if this VB.LIC licensing file exists when the control is loaded into the environment. If the file does not exist, or is corrupt, the control aborts the loading process and displays the following Alter message box:

> License file for custom control not found. You do not have an appropriate license to use this custom control in the design environment.

VB Graph Custom Control: DataReset Property Resets to 0 (Zero)

Article ID: Q84058

Summary:

When you assign a value to the DataReset property of the Graph version 1.2 custom control, the value of DataReset always resets to 0 - None. This is by design. Although DataReset is listed in the Properties box, it also has characteristics of a method. A value assigned to DataReset is transient, which means that it causes a one-time action and then resets to 0 - None.

More Information:

In Visual Basic for Windows, a property is an attribute of the control that you can set to define one of the object's characteristics. DataReset is a property because you can set its value which, depending on that value, defines one or more of the Graph control's characteristics. Because it defines a Graph's characteristics by resetting the chosen property array to its default values, DataReset is found in the Properties list box.

A method in Visual Basic for Windows behaves similarly to a statement in that it always acts on an object. DataReset can also be considered a method because it does perform an action on the graph. Namely, it resets the chosen property array to its default values. DataReset performs the assigned action as soon as its value does not equal 0. If it retained its assigned value, it would continually generate an endless loop and lock the system. To prevent this from occurring, it is automatically reset to 0 - None upon the first execution of its call.

The example below demonstrates the behavior of DataReset.

Example
----

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. From the File menu, choose Add File. In the Files box, select the GRAPH.VBX custom control file. The Graph tool will appear in the Toolbox.

3. Add a Graph control (Graph1) to Form1.

4. In the Properties list box, select the DataReset property. The value that appears in the Settings box will be 0 - None.

5. Change the value of DataReset to a number between 1 and 9. The values 1-9 refer to Graph property arrays that can be reset by using the DataReset property.

6. Graph1 will update to display the default values in the property array you chose in step 5.

7. In the Properties list box, select DataReset. The value of DataReset is 0 - None. It did not retain the value from step 5.

VB Graph Control: ThisPoint, ThisSet Reset to 1 at Run Time

Article ID: Q82155

Summary:

The Graph version 1.2 custom control in the Professional Edition of Microsoft Visual Basic versions 2.0 or 3.0 for Windows, and in the Microsoft Professional Toolkit for Visual Basic version 1.0 for Windows, allows you to set the values of the ThisPoint and ThisSet properties at design time to aid in the development of your graphs. However, when you run the project, the Graph custom control resets the property values of ThisPoint and ThisSet to 1.

This behavior is a design feature of the Graph custom control to help avoid logic errors in your code. If your program requires ThisPoint and ThisSet to be a value other than 1 upon execution of the project, you will need to specifically set these property values in the program's code.

More Information:

The example below demonstrates that ThisPoint and ThisSet are reset to 1 at run time.

Example
----

1. With Visual Basic for Windows running and Graph loaded, create a form (Form1).

2. On Form1 create a graph control (Graph1).

3. Change the following properties:

Control	Property	Value
Command1	Caption	Show values
Graph1	Top	2000
Graph1	NumSet	2
Graph1	ThisPoint	2
Graph1	ThisSet	2

4. Add the following code to the Command1 button Click event:

```
Sub Command1_Click ()
     Form1.Print "Graph1.ThisPoint = "; Graph1.ThisPoint
     Form1.Print "Graph1.ThisSet = "; Graph1.ThisSet
End Sub
```

5. Press the F5 key to run the program.

When you run the program and click on the Command1 button, the program will display the current values of Graph1.ThisPoint and Graph1.ThisSet. These values should have changed from 2 to 1.

VB Graph Control Displays Maximum of 80 Characters Per Title

Article ID: Q81450

Summary:

The Graph custom control has an 80 character maximum limit on all displayed strings such as labels and legends. However, the combined length of the actual string may be longer than 80 characters.

More Information:

The Graph custom control can display strings by using several different properties. For example, the BottomTitle and LeftTitle properties may be set from the Properties bar in the programming environment.

The following example sets the BottomTitle property of a Graph to 90 characters:

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. From the File menu, choose Add File. In the Files box, select the GRAPH.VBX custom control file. The Graph tool will appear in the toolbox.

3. Select the Graph icon on the toolbox and place it on Form1, and expand it to the largest size possible.

4. Double-click on the Graph control to open the Code window for the Click event.

5. Add the following code to the Click event:

```
Graph1.BottomTitle = String$(79, "i") + "*"
Debug.Print Len(Graph1.BottomTitle)
Graph1.DrawMode = 2          ' Update Graph.
```

6. Run the program and click on the graph control. If your Graph is expanded to the largest possible size, you should be able to see the string of 80 characters.

7. Change the code as follows:

```
Graph1.BottomTitle = String$(80, "i") + "*"
Debug.Print Len(Graph1.BottomTitle)
Graph1.DrawMode = 2               ' Update Graph.
```

You should not be able to see the last character, the asterisk (*).

In this example, 80 characters at most will show on the screen even though you set the BottomTitle property to a larger character string. The actual BottomTitle property, however, contains more characters. Whether or not the actual strings are displayed also depends on other factors, such as the width and height of the control, or the strings that are placed in the other properties of the control.

VB Key Status: Autosize Property Affects Height and Width

Article ID: Q81952

Summary:

In versions 2.0 or 3.0 of the Professional Edition of Microsoft Visual Basic for Windows, or in the Microsoft Professional Toolkit for Microsoft Visual Basic programming system version 1.0 for Windows, the Key Status control (KEYSTAT.VBX) allows you to show and set the current status of the CAPS LOCK, NUM LOCK, SCROLL LOCK and INSERT keys. One of the features of the Key Status control is its ability to size itself (the Autosize property) to its original dimensions. The property that affects this feature is Autosize. If Autosize is set to True (the default setting), the control's Height and Width properties will remain at, or be reset to its predetermined values. The size of the control cannot be changed if Autosize is set to True. If the Autosize property is set to False, the Height and Width properties can be changed to reflect the desired control size.

Note: Autosize can be set at both design time and run time.

VB MCI Control Does Not Support PC Speaker Driver

Article ID: Q84268

Summary:

The MCI custom does not support playing wave (.WAV) sound files through a PC speaker driver such as SPEAKER.DRV. The MCI custom control (and the Windows Media Player application) uses the MCI sound drivers, which do not support the PC speaker. The Windows default sounds and the Sound Recorder application are the only way to play sounds through the SPEAKER.DRV PC speaker driver.

More Information:

The MCI control manages the recording and playback of multimedia files on Media Control Interface (MCI) devices, such as audio boards, MIDI sequencers, CD-ROM drives, audio CD players, video disc players, and videotape recorders and players.

Although the MCI control will not allow you to play .WAV files through the PC speaker, you can use the Object Linked and Embedding (OLE) Client custom control provided with the Professional Edition of the Microsoft Visual Basic for Windows, or with the Microsoft Visual Basic for Windows Professional Toolkit to create and play a linked Sound Recorder object from your Visual Basic for Windows program. The following is an example of this behavior. (Note that you must have the appropriate Windows sound drivers loaded in order to run this program):

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. From the File menu, choose Add File. In the Files box, select the OLECLIEN.VBX custom control file. The OLE Client tool appears in the Toolbox.

3. Double-click on the OLE Client control on the tool bar to create an OLE Client control on your form.

4. Double-click on the form to open the Code window, and enter the following code in the Form_Click event:

```
OLEClient1.Class = "SoundRec"
OLEClient1.Protocol = "StdFileEditing"
OLEClient1.SourceDoc = "C:\windows\chimes.wav"   ' Name of .WAV file.
OLEClient1.SourceItem = "LINK"
OLEClient1.ServerType = 0      ' Linked object.

OLEClient1.Action = 1      ' Create object from source file.
OLEClient1.Action = 7      ' Activate Sound Recorder - plays sound.
OLEClient1.Action = 10     ' Delete the object.
```

5. Press the F5 key to run the program.

The specified .WAV file should be played each time you click on the form.

VB MCI Control Does Not Support Recording of MIDI Data

Article ID: Q84473

Summary:

The Multimedia Device control called MCI (MCI.VBX), consists of a set of high level, device-independent commands that control audio and visual peripherals. However, the MCI control cannot record standard MIDI (Musical Instrument Data Interface) input. This is a limitation of the MCI control, not of Visual Basic for Windows.

Below is an example of using the MCI control to play back a MIDI file.

More Information:

The MCI custom control can play back MIDI files if you have the necessary hardware and software installed. Typically, you need a sound board that supports MIDI and Windows, version 3.1 to use the MCI control to play back MIDI files. Windows 3.1 or (Windows 3.0 with Multimedia Extensions version 1.0) supplies MIDI drivers for several well-known hardware add-on boards that support MIDI.

The following is an example of using the MCI control to play back a MIDI file called TEST.MID.

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. From the File menu, choose Add File. In the Files box, select the MCI.VBX custom control file. The MCI tool appears in the Toolbox.

3. Add the following code to the Form_Load event procedure:

```
Sub Form_Load ()
  MMControl1.PlayVisible = -1
  MMControl1.StopVisible = -1
  MMControl1.FileName = "c:\midi\bach.mid"
  MMControl1.Wait = -1
  MMControl1.DeviceType = "sequencer"
  MMControl1.Command = "open"
End Sub
```

4. Add the following code to your Form_Unload event procedure:

```
Sub Form_Unload (Cancel As Integer)
  MMControl1.Command = "close"
End Sub
```

5. Press the F5 key to run the program. Click on the play arrow of the MCI control to play the MIDI file.

Note: An MIDI file may play, but may not be audible due to MIDI configuration issues such as the MIDI channel and instrument.

How to Close VB Combo Box with ENTER key

Article ID: Q84474

Summary:

If you open a combo box and then use the ARROW keys to scroll through it, pressing the ENTER key will not close the combo box like a mouse click will. This is normal behavior. The following example demonstrates how to make a combo box close when the ENTER key is pressed.

More Information:

The following program makes use of the Windows API SendMessage function to send the combo box the message to close. This is done only after the ENTER key is detected in the KeyPress event for the combo box.

Two Windows API Declare statements must be added to your application. These can be added either in the GLOBAL.BAS module, or in the general Declarations section of the form containing the combo box.

Steps to Reproduce Behavior
---

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Add the following two declarations to the global module or the General Declarations for Form1:

   ```
   Declare Function SendMessage% Lib "user" (ByVal hWnd%, ByVal
                  wMsg%, ByVal wParam%, ByVal lParam&)
   Declare Function GetFocus Lib "user" () As Integer
   ```

   (Note that the first Declare statement must be on just one line, not split across two lines as it is here.)

3. Place a combo box on Form1.

4. Under the KeyPress event for the combo box, place the following code:

   ```
    If KeyAscii = 13 Then
        Const WM_USER = &h400
        Const  CB_SHOWDROPDOWN = WM_USER + 15

        Combo1.SetFocus
        BoxwHND% = GetFocus()
        r& = SendMessage(BoxwHND%, CB_SHOWDROPDOWN, 0, 0)
        KeyAscii = 0
    End If
   ```

5. Place a command button on Form1.

6. In the Click event for Command1, place the following code:

```
' This will add some data to the combo box.
for i =1 to 10
   Combo1.AddItem STR$(i)
Next i
```

7. Press the F5 key to run the application.

8. Choose the Command1 button to fill the combo box.

9. Open the combo box with the mouse, and scroll down with the ARROW keys. Pressing the ENTER key will close the Combo Box.

How to Limit User Input in VB Combo Box with SendMessage API

Article ID: Q72677

Summary:

You can specify a limit to the amount of text that can be entered into a combo box by calling SendMessage (a Windows API function) with the EM_LIMITTEXT constant.

More Information:

The following method can be used to limit the length of a string entered into a combo box. Check the length of a string inside a KeyPress event for the control, if the length is over a specified amount, then the formal argument parameter KeyAscii will be set to zero.

Or, the preferred method of performing this type of functionality is to use the SendMessage API function call. After you set the focus to the desired edit control, you must send a message to the window's message queue that will reset the text limit for the control. The argument EM_LIMITTEXT, as the second parameter to SendMessage, will set the desired text limit based on the value specified by the third arguments. The SendMessage function requires the following parameters for setting the text limit:

```
SendMessage (hWnd%,EM_LIMITTEXT, wParam%, lParam)

wParam%   Specifies the maximum number of bytes that can be
          entered. If the user attempts to enter more characters,
          the edit control beeps and does not accept the characters.
          If the wParam parameter is zero, no limit is imposed on
          the size of the text (until no more memory is available).

lParam    Is not used.
```

The following steps can be used to implement this method:

1. Create a form called Form1.

2. Add a combo box called Combo1 to Form1.

3. Add the following code to the general declarations section of Form1:

```
'*** Note: Each Declare statement must be on just one line:

Declare Function GetFocus% Lib "user" ()
Declare Function SendMessage& Lib "user" (ByVal hWnd%,
                                          ByVal wMsg%,
                                          ByVal wParam%,
                                          lp As Any)
Const WM_USER = &H400
Const EM_LIMITTEXT = WM_USER + 21
```

4. Add the following code to the Form_Load event procedure:

```
Sub Form_Load ()
   Form1.Show            ' Must show form to work on it.
   Combo1.SetFocus       ' Set the focus to the list box.
```

```
    cbhWnd% = GetFocus()  ' Get the handle to the list box.
    TextLimit% = 5        ' Specify the largest string.
    retVal& = SendMessage(cbhWnd%, EM_LIMITTEXT, TextLimit%, 0)
End Sub
```

5. Run the program and enter some text into the combo box. You will notice that you will only be able to enter a string of five characters into the combo box.

DEL Key Behavior Depends on Text Box MultiLine Property

Article ID: Q77737

Summary:

Pressing the DEL key in a multiline text box generates a KeyPress event for that text box with an ASCII code of 8 for the key. In a standard text box, no KeyPress event is generated for the DEL key. This behavior is inherent to Windows and is not specific to Microsoft Visual Basic for Windows.

More Information:

Steps to Reproduce Problem
---

1. Place a text box on a form.

2. Set the MultiLine property for the text box to True.

3. Add the following code to the text box KeyPress event:

```
Sub Text1_KeyPress (keyAscii as Integer)
     debug.print keyAscii  ' This will print the generated ASCII
                           ' code to VB's Immediate window.
End Sub
```

4. Execute the program and press the DEL key while the focus is on the text box. An "8" will be printed in the Immediate window.

If the text box's MultiLine property is set to false, no KeyPress event occurs and nothing is printed to the Immediate window when you press the DEL key. This behavior is standard for Windows multiline text boxes.

Determining Number of Lines in VB Text Box; SendMessage API

Article ID: Q72719

Summary:

To determine the number of lines of text within a text box control, call the Windows API function SendMessage with EM_GETLINECOUNT(&H40A) as the wMsg argument.

Calling SendMessage with the following parameters will return the amount of lines of text within a text box:

```
hWd%   - Handle to the text box.
wMsg%   - EM_GETLINECOUNT(&H40A)
wParam% - 0
lParam% - 0
```

More information:

For example, to determine the amount of lines within a text box, perform the following steps:

1. Create a form with a text box and a command button. Change the MultiLine property of the text box to TRUE.

2. Declare the API SendMessage function in the global-declarations section of your code window (the Declare statement must be on just one line):

```
Declare Function SendMessage% Lib "user" (ByVal hWd%,
                                          ByVal wMsg%,
                                          ByVal wParam%,
                                          ByVal lParam&)
```

3. In Visual Basic version 1.0 for Windows, you will need to declare another API routine to get the handle of the text box. Declare this routine also in your global declarations section of your code window. The returned value will become the hWd% argument to the SendMessage function. For example:

```
Declare Function GetFocus% Lib "user" ()
```

4. Within the click event of your button, add the following code:

```
Sub Command1_Click ()
   Const EM_GETLINECOUNT = &H40A  ' Defined within Windows SDK
                                  ' file, WINDOWS.H.

   ' Command button has focus, give focus to text box.
   Text1.SetFocus

   ' For Visual Basic 1.0 for Windows get the handle of the text box.
   ' hWd% = GetFocus()

   ' Print the amount of lines to the immediate window.
   Debug.Print SendMessage(Text1.hWnd, EM_GETLINECOUNT, 0, 0)
   ' For Visual Basic 1.0 for Windows use hWd% instead of Text1.hWnd.
End Sub
```

5. Run the program. Add several lines of text to the text box. Click the command button to see the number of lines printed out to the immediate window.

Disabling the ENTER Key BEEP in a Visual Basic Text Box

Article ID: Q78305

Summary:

In a Microsoft Visual Basic for Windows text box, the ENTER key causes a warning beep to sound only if the MultiLine property is set to False (the default) and the Warning Beep option is selected in the Sound dialog box of the Windows Control panel. To disable the beep, in the KeyPress event procedure for the text box, set the value of KeyAscii (which is a parameter passed to KeyPress) equal to zero (0) when the user presses the ENTER key.

More Information:

Specifically, use an IF statement to trap the ENTER key and the set KeyAscii to zero (0). Setting the value to zero before the event procedure ends prevents Windows from detecting that the ENTER key was pressed and prevents the warning beep. This behavior is by design and is due to the fact that a non-multiline text box is a Windows default class of edit box.

Example

The following code will prevent the beep.

```
' (Set Multiline property to False).

Sub Text1_KeyPress (KeyAscii as Integer)
  If KeyAscii=13 Then
    KeyAscii=0
  End If
End Sub
```

How to Scroll VB Text Box Programmatically and Specify Lines

Article ID: Q73371

Summary:

By making a call to the Windows API function SendMessage, you can scroll text a specified number of lines or columns within a Microsoft Visual Basic for Windows text box. By using SendMessage, you can also scroll text programmatically, without user interaction. This technique extends Visual Basic for Windows' scrolling functionality beyond the built-in statements and methods. The sample program below shows how to scroll text vertically and horizontally a specified number of lines.

More Information:

Note that Visual Basic for Windows itself does not offer a statement for scrolling text a specified number of lines vertically or horizontally within a text box. You can scroll text vertically or horizontally by actively clicking on the vertical and horizontal scroll bars for the text box at run time; however, you do not have any control over how many lines or columns are scrolled for each click of the scroll bar. Text always scrolls one line or one column per click on the scroll bar. Furthermore, no built-in Visual Basic for Windows method can scroll text without user interaction. To work around these limitations, you can call the Windows API function SendMessage, as explained below.

Example

To scroll the text a specified number of lines within a text box requires a call to the Windows API function SendMessage using the constant EM_LINESCROLL. You can invoke the SendMessage function from Visual Basic for Windows as follows:

```
r& = SendMessage& (hWd%, EM_LINESCROLL, wParam%, lParam&)
```

hWd%	The window handle of the text box.
wParam%	Parameter not used.
lParam&	The low-order 2 bytes specify the number of vertical lines to scroll. The high-order 2 bytes specify the number of horizontal columns to scroll. A positive value for lParam& causes text to scroll upward or to the left. A negative value causes text to scroll downward or to the right.
r&	Indicates the number of lines actually scrolled.

The SendMessage API function requires the window handle (hWd% above) of the text box. To get the window handle of the text box, you must first set the focus on the text box using the SetFocus method from Visual Basic. Once the focus has been set, call the GetFocus API function to get the window handle for the text box. Below is an example of how to get the window handle of a text box.

```
' The following appears in the general declarations section of
' the form:
Declare Function GetFocus% Lib "USER" ()

' Assume the following appears in the click event procedure of a
```

```
' command button called Scroll.
Sub Command_Scroll_Click ()
     OldhWnd% =  Screen.ActiveControl.Hwnd
     ' Store the window handle of the control that currently
     ' has the focus.

     ' For Visual Basic 1.0 for Windows use the following line:
     ' OldhWnd% =  GetFocus ()

     Text1.SetFocus
     hWd% = GetFocus()
End Sub
```

To scroll text horizontally, the text box must have a horizontal scroll bar, and the width of the text must be wider than the text box width. Calling SendMessage to scroll text vertically does not require a vertical scroll bar, but the length of text within the text box should exceed the text box height.

Below are the steps necessary to create a text box that will scroll five vertical lines or five horizontal columns each time you click the command buttons labeled "Vertical" and "Horizontal":

1. From the File menu, choose New Project (press ALT, F, N).

2. Double-click on Form1 to bring up the code window.

3. Add the following API declaration to the General Declarations section of Form1. Note that you must put all Declare statements on a separate and single line. Also note that SetFocus is aliased as PutFocus because there already exists a SetFocus method within Visual Basic for Windows.

   ```
   Declare Function GetFocus% Lib "user" () ' For Visual Basic 1.0 only.
   Declare Function PutFocus% Lib "user" Alias "SetFocus" (ByVal
                                                           hWd%)
   Declare Function SendMessage& Lib "user" (ByVal hWd%,
                                             ByVal wMsg%,
                                             ByVal wParam%,
                                             ByVal lParam&)
   ```

4. Create a text box called Text1 on Form1. Set the MultiLine property to True and the ScrollBars property to Horizontal (1).

5. Create a command button called Command1 and change the Caption to "Vertical".

6. Create a another command button called Command2 and change the Caption to "Horizontal".

7. From the General Declarations section of Form1, create a procedure to initialize some text in the text box as follows:

   ```
   Sub InitializeTextBox ()
      Text1.Text = ""
      For i% = 1 To 50
         Text1.Text = Text1.Text + "This is line " + Str$(i%)
   ```

```
      ' Add 15 words to a line of text.
      For j% = 1 to 10
         Text1.Text = Text1.Text + " Word "+ Str$(j%)
      Next j%

      ' Force a carriage return (CR) and linefeed (LF).
      Text1.Text = Text1.Text + Chr$(13) + Chr$(10)

      x% = DoEvents()
   Next i%
End Sub
```

8. Add the following code to the load event procedure of Form1:

```
Sub Form_Load ()
   Call InitializeTextBox
End Sub
```

9. Create the actual scroll procedure within the General Declarations section of Form1 as follows:

```
' The following two lines must appear on a single line:
Function ScrollText& (TextBox As Control, vLines As Integer, hLines
                      As Integer)
   Const EM_LINESCROLL = &H406

   ' Place the number of horizontal columns to scroll in the high-
   ' order 2 bytes of Lines&. The vertical lines to scroll is
   ' placed in the low-order 2 bytes.
   Lines& = Clng(&H10000 * hLines) + vLines

   ' Get the window handle of the control that currently has the
   ' focus, Command1 or Command2.
   SavedWnd%  = Screen.ActiveControl.Hwnd
   ' For Visual Basic 1.0 use the following line instead of the one
   ' used above.
   ' SavedWnd% = GetFocus%()

   ' Set the focus to the passed control (text control).
   TextBox.SetFocus

   ' For Visual Basic 1.0, get the handle to current focus (text
   ' control).
   ' TextWnd% = GetFocus%()

   ' Scroll the lines.
   Success& = SendMessage(TextBox.HWnd, EM_LINESCROLL, 0, Lines&)
   ' For Visual Basic 1.0 use the following line instead of the one
   ' used above.
   ' Success& = SendMessage(TextWnd%, EM_LINESCROLL, 0, Lines&)

   ' Restore the focus to the original control, Command1 or
   ' Command2.
   r% = PutFocus% (SavedWnd%)

   ' Return the number of lines actually scrolled.
```

```
   ScrollText& = Success&

End Function
```

10. Add the following code to the click event procedure of Command1 labeled "Vertical":

```
Sub Command1_Click ()
   ' Scroll text 5 vertical lines upward.
   Num& = ScrollText&(Text1, 5, 0)
End Sub
```

11. Add the following code to the click event procedure of Command2 labeled "Horizontal":

```
Sub Command2_Click ()
    ' Scroll text 5 horizontal columns to the left.
    Num& = ScrollText&(Text1, 0, 5)
End Sub
```

12. Run the program. Click the command buttons to scroll the text five lines or columns at a time.

UCase$/LCase$ in Text Box Change Event Inverts Text Property

Article ID: Q84059

Summary:

When using the UCase$ or LCase$ functions in Microsoft Visual Basic for Windows to capitalize text or make text lower case from within the change procedure of a text box, you may encounter unexpected results if the following conditions are true:

- The text property of the text box is being updated by the UCase$ or LCase$ statement.
- The resulting string created by UCase$ or LCase$ is assigned to the text property of the text box.
- The above statements appear in the Change event procedure of the text box.

Every time a key is pressed, the text contents are changed, and the cursor is placed at the beginning of the line. This causes the character for your next key press to be inserted at the beginning of the line rather than the end.

More Information:

When allowing users to enter text into text boxes, it is often desirable to control whether the user enters all uppercase or all lowercase letters. To do this, it would seem that putting a UCase$ or LCase$ statement in a text box Change event would allow you to enter only uppercase or lowercase letters into the text box. However, each time you press a key, the Change event fires and the cursor is brought back to the beginning of the text box as a result of assigning the Text property a new string.

Steps to Reproduce Behavior
---

1. Start Visual Basic for Windows or from the File menu, select New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Put a text box (Text1) on Form1 by either double-clicking the text box control or single clicking on the text box control and drawing it on Form1.

3. Add the following code to the Text1_Change event procedure:

```
Sub Text1_Change ()
  text1.text = UCase$(text1.text)
End Sub
```

4. Press the F5 key to run the program.

Notice that when you try to type information into the text box that it is entered in reverse order of what you would expect.

An alternative method of changing all contents of the text box to

capital letters is to change the KeyAscii code of the typed information in the text box KeyPress event as follows:

```
Sub Text1_KeyPress (KeyAscii As Integer)

' Check to see if key pressed is a lower case letter.
  If KeyAscii >= 97 And KeyAscii <= 122 Then

    'If it is lowercase, change it to uppercase.
    KeyAscii = KeyAscii - 32

  End If

End Sub
```

When you run the above code, the letters typed into the text box are immediately changed to capital letters and are entered correctly as you type them in.

Another alternative method of changing the contents of the text box to uppercase letters is to add the following code to the Change event for the text box:

```
Sub Text1_Change ()

' Get the current position of the cursor.
  CurrStart = Text1.SelStart

' Change the text to capitals.
  Text1.Text = UCase$(Text1.Text)

' Reset the cursor position.
  Text1.SelStart = CurrStart

End Sub
```

SelStart sets or returns the starting point of text selected, and indicates the position of the insertion point if no text is selected.

VB Can Call Escape API to Specify Number of Copies to Printer

Article ID: Q78165

Summary:

You can call the Windows API Escape function to tell the Windows Print Manager how many copies of a document you want to print.

More Information:

The Windows API constant SETCOPYCOUNT (value 17) can be used as an argument to the Escape function to specify the number of uncollated copies of each page for the printer to print.

The arguments for Escape are as follows:

```
r% = Escape(hDC, SETCOPYCOUNT, Len(Integer), lpNumCopies, lpActualCopies)
```

Parameter	Type/Description
hDC	hDC. Identifies the device context. Usually referenced by Printer.hDC.
lpNumCopies	Long pointer to integer (not ByVal). Point to a short-integer value that contains the number of uncollated copies to print.
lpActualCopies	Long pointer to integer (not ByVal). Points to a short integer value that will receive the number of copies that where printed. This may be less than the number requested if the requested number is greater than the device's maximum copy count.

The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is a negative number if the escape is not successful. If the escape is not supported, the return value is zero.

The following sample will demonstrate how to print three copies of a line of text to the printer. To recreate this example, create a new project from the Visual Basic File menu and add a command button. Paste the following code into the appropriate code sections of your program:

```
REM   Below is GLOBAL.BAS:

' Note: the following Declare must be on one line:
Declare Function Escape% Lib "GDI" (
        ByVal hDc%,
        ByVal nEsc%,
        ByVal nLen%,
        lpData%,
        lpOut%)

REM   Below is the click procedure for a command button on FORM1:

Sub Command1_Click ()
  Const SETCOPYCOUNT = 17
```

```
     Printer.Print ""
     x% = Escape(Printer.hDC, SETCOPYCOUNT, Len(I%), 3, actual%)
     Printer.Print " Printing three copies of this"
     Printer.EndDoc
End Sub
```

How to Set Landscape or Portrait for Printer in Windows 3.0

Article ID: Q80185

Summary:

Some printers support changing the orientation of the paper output to landscape. With the Windows 3.0 API Escape function, you can change the settings of the printer to either landscape or portrait.

Below is an example of invoking the Windows 3.0 API Escape function from Microsoft Visual Basic programming system version 1.0 for Windows.

Important Note: The Windows API Escape function used below is provided in Windows 3.0 only for backward compatibility with earlier Microsoft Windows releases. New applications should use the GDI DeviceCapabilities and ExtDeviceMode functions instead of the Escape function shown below.

More Information:

Normally, output for the printer is in portrait mode, where output is printed horizontally across the narrower dimension of a paper. In landscape mode, the output is printed horizontally across the longer dimension of the paper.

You can use the Escape function to change the orientation of the printer by passing GETSETPAPERORIENT as an argument. When you initially print text to the printer, Visual Basic will use the currently selected orientation. Sending the Escape function will not take effect until you perform a Printer.EndDoc. After you perform a Printer.EndDoc, output will print in the orientation that you have selected.

To determine if your printer supports landscape mode, do the following:

1. From the Windows 3.0 Program Manager, run Control Panel.

2. From the Control Panel, select the Printers icon.

3. From the Printers dialog box, choose the Configure button.

4. The Configure dialog box will contain an option for landscape orientation if landscape is supported on your printer.

The example below demonstrates how to change the printer orientation to landscape. Please note that your printer must support landscape mode for these commands to have any effect.

Code Example
---

1. Run Visual Basic, or from the File menu, choose New Project (ALT, F, N) if Visual Basic is already running. Form1 is created by default.

2. Add a command button (Command1) to Form1.

3. Add the following code to the global module:

```
GLOBAL.BAS
----------
Type OrientStructure
  Orientation As Long
  Pad As String * 16
End Type
' The following Declare statement must be on a single line:
Declare Function Escape% Lib "GDI" (ByVal hDc%, ByVal nEsc%,
               ByVal nLen%, lpData As OrientStructure, lpOut As Any)
```

4. Add the following code to the Command1_Click event procedure of the Command1 button:

```
FORM1
-----
Sub Command1_Click ()
  Const PORTRAIT = 1
  Const LANDSCAPE = 2
  Const GETSETPAPERORIENT = 30
  Const NULL = 0&

  Dim Orient As OrientStructure

    '* Start the printer
    Printer.Print ""

    '* Specify the orientation
    Orient.Orientation = LANDSCAPE

    '* Send escape sequence to change orientation
    x% = Escape(Printer.hDC, GETSETPAPERORIENT,
                Len(Orient), Orient, NULL)
    '* The EndDoc will now re-initialize the printer
    Printer.EndDoc

    Printer.Print "Should print in landscape mode"
    Printer.EndDoc
End Sub
```

Using PASSTHROUGH Escape to Send Data Directly to Printer

Article ID: Q96795

Summary:

By using the Windows API Escape() function, your application can pass data directly to the printer. If the printer driver supports the PASSTHROUGH printer escape, you can use the Escape() function and the PASSTHROUGH printer escape to send native printer language codes to the printer driver.

Printer escapes such as PASSTHROUGH allow applications to access certain facilities of output devices that are not directly available through the graphics device interface (GDI). The PASSTHROUGH printer escape allows the application to send data directly to the printer, bypassing the standard print-driver code.

More Information:

A printer driver that supports the PASSTHROUGH printer escape does not add native printer language codes to the data stream sent to the printer, so you can send data directly to the printer. However, Microsoft recommends that applications not perform functions that consume printer memory, such as downloading a font or a macro.

The sample program listed below sends native PCL codes to the printer to change the page orientation and the paper bin. A Hewlett-Packard LaserJet is the assumed default printer.

An Important Note

The Windows API Escape() function used below is provided in Windows version 3.0 and later for backward compatibility with earlier versions of Microsoft Windows. New applications should use the GDI DeviceCapabilities() and ExtDeviceMode() functions instead of the Escape() function shown below. In order to use the ExtDeviceMode() function, you need to create a DLL because the ExtDeviceMode() function is exported by the printer driver, not by the Windows GDI. To execute the ExtDeviceMode() function, you need to obtain a function pointer to it from the current printer driver. Visual Basic does not support pointers.

Steps to Create Example
---

1. Start Visual Basic or from the File menu, choose New Project (ALT, F, N) if Visual Basic is already running. Form1 is created by default.

2. Add the following code to the general declarations section of Form1:

```
' Enter the entire Declare statement on one, single line.
Declare Function Escape Lib "Gdi" (ByVal Hdc%, ByVal nEscape%,
   ByVal ncount%, ByVal indata$, ByVal oudata$) As Integer

Const PASSTHROUGH = 19

Const RevLandScape = "&l3O" ' PCL command to change Paper
                            ' orientation to Reverse Landscape.
Const Portrait = "&l0O"     ' PCL command to change paper
                            ' orientation to Portrait.
Const ManualFeed = "&l3H"   ' PCL command to change Paper Bin
```

```
                                ' to Manual Feed Envelope.
   Const AutoFeed = "&l1H"      ' PCL command to change Paper Bin
                                ' to Paper Tray AutoFeed
```

3. Add a list box (List1) to Form1.

4. Add the following code to Form1's Form_Load event procedure:

```
   Sub Form_Load ()
     List1.AddItem "HP/PCL Reverse Landscape"
     List1.AddItem "HP/PCL Portrait"
     List1.AddItem "HP/PCL Manual Feed Envelope"
     List1.AddItem "HP/PCL Paper Tray Auto Feed"
   End Sub
```

5. Add the following code to the List1_Click event procedure:

```
   Sub List1_Click
   Select Case List1.ListIndex
      Case 0:
         PCL_Escape$ = Chr$(27) + RevLandScape
      Case 1:
         PCL_Escape$ = Chr$(27) + Portrait
      Case 2:
         PCL_Escape$ = Chr$(27) + ManualFeed
      Case 3:
         PCL_Escape$ = Chr$(27) + AutoFeed
   End Select

   PCL_Escape$ = Chr$(Len(PCL_Escape$)) + PCL_Escape$ + Chr$(0)

   ' Enter the entire Result% statement on one, single line.
   Result% = Escape%(Printer.hDC, PASSTHROUGH, Len(PCL_Escape$),
       PCL_Escape$, "")

   Select Case Result%
      ' Enter each Case statement on one, single line.
      Case Is < 0: MsgBox "The PASSTHROUGH Escape is not
         supported by this printer driver.", 48
      Case 0: MsgBox "An error occurred sending the escape
         sequence.", 48
      Case Is > 0: MsgBox "Escape Successfully sent.
         Sending test printout to printer."
      Printer.Print "Test case of "; List1.Text
   End Select
   End Sub
```

6. From the Run menu, choose Start (ALT, R, S) to run the program. List1 is filled with four escape sequences to send to the printer.

7. Select any of the options in the list box. A message box appears to indicate the success of the operation.

If the printer driver does not support the PASSTHROUGH printer escape, you must use the DeviceCapabilities() and ExtDevMode() functions instead.

Using an Escape to Obtain and Change Paper Size for Printer

Article ID: Q96796

Summary:

By using the Windows API Escape() function, an application can change the paper size on the printer and obtain a list of available paper metrics for the default printer.

To get the list of available paper metrics, pass the ENUMPAPERMETRICS printer escape constant to the Escape() function. The function will return either an array containing the paper metrics or the number of paper metrics available. Note that paper metrics differ from the physical paper sizes in that paper metrics delineate the actual region that can be printed to, whereas paper size is the physical size of the paper including the non-printable regions.

To change the paper size, pass the GETSETPAPERMETRICS printer escape constant along with the paper metrics to the Escape() function.

More Information:

The example program listed below demonstrates how to use both printer escape constants (ENUMPAPERMETRICS and GETSETPAPERMETRICS) with the Windows API Escape() function.

An Important Note

The Windows API Escape() function used below is provided in Windows version 3.0 and later for backward compatibility with earlier versions of Microsoft Windows. New applications should use the GDI DeviceCapabilities() and ExtDeviceMode() functions instead of the Escape() function shown below. In order to use the ExtDeviceMode() function, you need to create a DLL because the ExtDeviceMode() function is exported by the printer driver, not by the Windows GDI. To execute the ExtDeviceMode() function, you need to obtain a function pointer to it from the current printer driver. Visual Basic does not support pointers.

Steps to Create Example:
---
1. Start Visual Basic or from the File menu, choose New Project (ALT, F, N) if Visual Basic is already running. Form1 is created by default.

2. From the File menu, choose New Module (ALT, F, M). Module1 is created by default.

3. Add the following code to the general declarations section of Module1:

```
Type Rect
   Left As Integer
   Top As Integer
   Right As Integer
   Bottom As Integer
End Type

' Enter each Declare as one, single line.
Declare Function EnumPaperMetricsEscape% Lib "GDI" Alias "Escape"
   (ByVal hDC%, ByVal nEscape%, ByVal IntegerSize%, lpMode%,
```

```
   lpOutData As Rect)
Declare Function SetPaperMetricsEscape% Lib "GDI" Alias "Escape"
   (ByVal hDC%, ByVal nEscape%, ByVal RectSize%, NewPaper As Rect,
   PrevPaper As Rect)
Declare Function GetDeviceCaps% Lib "gdi" (ByVal hDC%, ByVal nIndex%)

Global Const ENUMPAPERMETRICS = 34
Global Const GETSETPAPERMETRICS = 35
Global Const LOGPIXELSX = 88    '  Logical pixels/inch in X
Global Const LOGPIXELSY = 90    '  Logical pixels/inch in Y
```

4. Add the following code to the General Declarations section of Form1:

```
Dim RectArray() As Rect
```

5. Add a command button (Command1) to Form1.

6. Add a list box (List1) to Form1.

7. Add the following code to the Command1_Click event procedure. For readability some lines of code are shown as two lines but must be entered as a single line of code.

```
Sub Command1_Click ()
   ReDim RectArray(1)
   mode% = 0
   ' Enter the entire Result% statement as one, single line.
   Result% = EnumPaperMetricsEscape(Printer.hDC, ENUMPAPERMETRICS,
      2, mode%, RectArray(0))
   If Result% = 0 Then    ' If Result = 0, the call failed
      MsgBox "Printer Driver does not Support EnumPaperMetrics", 48
      Command1.Enabled = False
      Exit Sub
   End If

   ReDim RectArray(Result% - 1) ' Result% contains num paper sizes
   mode% = 1
   ' Enter the entire Result2% statement as one, single line.
   Result2% = EnumPaperMetricsEscape(Printer.hDC, ENUMPAPERMETRICS,
      2, mode%, RectArray(0))
   HorzRatio% = GetDeviceCaps(Printer.hDC, LOGPIXELSX)
   VertRatio% = GetDeviceCaps(Printer.hDC, LOGPIXELSY)

   ' Add Paper Sizes (Listed by actual printing region) in inches
   ' to the list box. Enter each of the PWidth$ and PHeight$ statements
   ' as one, single line.
   For i% = 0 To Result% - 1
      PWidth$ = Format$((RectArray(i%).Right - RectArray(i%).Left)
         / HorzRatio%) + Chr$(34) ' Enter as a single line
      PHeight$ = Format$((RectArray(i%).Bottom - RectArray(i%).Top)
         / VertRatio%) + Chr$(34) ' Enter as a single line
      List1.AddItem PWidth$ + " X " + PHeight$
   Next i%
End Sub
```

8. Add the following code to the List1_Click event procedure:

```
Sub List1_Click ()
   Dim PrevPaperSize As Rect
   ' Enter the entire Result% statement as one, single line.
   Result% = SetPaperMetricsEscape(Printer.hDC, GETSETPAPERMETRICS,
      Len(PrevPaperSize), RectArray(List1.ListIndex), PrevPaperSize)

   If Result% = 0 Then
      MsgBox "Printer Driver does not support this Escape.", 48
   ElseIf Result% < 0 Then
      MsgBox "Error in calling Escape with GETSETPAPERMETRICS."
   Else
      MsgBox "Paper size successfully changed!"
   End If
End Sub
```

9. From the Run menu, choose Start (ALT, R, S) to run the program.

10. Choose the Command1 button to display a list of available paper metrics in the List1 box. The paper metrics represent the size of the printable regions supported by the printer, not the physical paper sizes.

11. Select one of the paper metrics shown in the List1 box. A message box appears indicating whether or not the paper size was successfully changed using the paper metrics you selected.

How to Obtain & Change the Paper Bins for the Default Printer

Article ID: Q96797

Summary:

By using the Windows API Escape() function, an application can change the paper bin on the printer and obtain a list of available paper bins for the default printer.

To return a list of paper bin names and a list of corresponding of bin numbers, pass the ENUMPAPERBINS printer escape constant to the Escape() function. You can use the first list to display the available paper bins for the user, and use the second list to change the paper bin.

To change the paper bin, pass the GETSETPAPERBINS printer escape constant along with the bin number to the Escape() function. GETSETPAPERBINS returns the current bin and the number of bins supported by the default printer.

More Information:

The example code listed below demonstrates how to use both ENUMPAPERBINS and GETSETPAPERBINS with the Windows API Escape() function.

An Important Note

The Windows API Escape() function used below is provided in Windows version 3.0 and later for backward compatibility with earlier versions of Microsoft Windows. New applications should use the GDI DeviceCapabilities() and ExtDeviceMode() functions instead of the Escape() function shown below. In order to use the ExtDeviceMode() function, you need to create a DLL because the ExtDeviceMode() function is exported by the printer driver, not by the Windows GDI. To execute the ExtDeviceMode() function, you need to obtain a function pointer to it from the current printer driver. Visual Basic does not support pointers.

Steps to Create Example:
---

1. Start Visual Basic or from the File menu, choose New Project (ALT, F, N) if Visual Basic is already running. Form1 is created by default.

2. From the File menu, choose New Module (ALT, F, M). Module1 is created by default.

3. Add the following code to the general declarations section of Module1:

```
Global Const MaxBins = 6
Type PaperBin          ' Used for EnumPaperBins
   BinList(1 To MaxBins) As Integer
   PaperNames(1 To MaxBins) As String * 24
End Type

Type BinInfo                ' Used for GetSetPaperBins
   CurBinNumber As Integer  ' Current Bin
   NumBins As Integer       ' Number of bins supported by printer
   Reserved1 As Integer     ' Reserved
   Reserved2 As Integer     ' Reserved
   Reserved3 As Integer     ' Reserved
   Reserved4 As Integer     ' Reserved
```

```
End Type
' Enter each complete Declare statement on one, single line.
Declare Function EnumPaperBinEscape% Lib "GDI" Alias "Escape"
   (ByVal hDC%, ByVal nEscape%, ByVal nCount%, NumBins%,
   lpOutData As Any)
Declare Function GetPaperBinEscape% Lib "GDI" Alias "Escape"
   (ByVal hDC%, ByVal nEscape%, ByVal nCount%, InBinInfo As Any,
   OutBinInfo As Any)

Global Const ENUMPAPERBINS = 31
Global Const GETSETPAPERBINS = 29
```

4. Add a command button (Command1) to Form1.

5. Add a list box (List1) to Form1.

6. Add the following code to the Command1_Click event procedure:

```
Sub Command1_Click ()
   Dim InPaperBin As PaperBin
   Dim InBinInfo As BinInfo
   ' Enter each complete result% statement on one, single line.
   result% = GetPaperBinEscape(Printer.hDC, GETSETPAPERBINS, 0,
      ByVal 0&, InBinInfo)
    result% = EnumPaperBinEscape(Printer.hDC, ENUMPAPERBINS, 2,
      MaxBins, InPaperBin)

   List1.Clear
   For I% = 1 To InBinInfo.NumBins ' Fill list1 with available bins
      List1.AddItem InPaperBin.PaperNames(I%)
      List1.ItemData(List1.NewIndex) = InPaperBin.BinList(I%)
   Next I%

End Sub
```

7. Add the following code to the List1_Click event procedure:

```
Sub List1_Click ()
   Dim InBinInfo As BinInfo
   Dim NewBinInfo As BinInfo

   NewBinInfo.CurBinNumber = List1.ItemData(List1.ListIndex)
   ' Enter the following result% statement on one, single line.
   result% = GetPaperBinEscape(Printer.hDC, GETSETPAPERBINS,
      Len(NewBinInfo), NewBinInfo, PrevBinInfo)

   MsgBox "Sending Sample Output to printer using Bin: " + List1.Text
   Printer.Print "This should of have come from Bin: "; List1.Text
   Printer.EndDoc
End Sub
```

8. From the Run menu, choose Start (ALT, R, S) to run the program.

9. Choose the Command1 button to see a list of available paper bins for the default printer listed in the List1 box.

10. Select one of the paper bins listed in the List1 box. A message box

appears to tell you that a sample printout is being sent to the printer using the paper bin you selected.

How to Use More than One Type of Font in Picture Box

Article ID: Q81220

Summary:

The text box control in Visual Basic for Windows displays the entire text box with either the FontUnderline, FontBold, FontItalic, or FontStrikethru fonts, but with only one font at a time. This behavior is by design.

However, you may want to display a box with all four fonts at the same time with separate words displayed in different fonts. Below is an example of displaying the fonts FontBold, FontItalic, FontStrikethru, and FontUnderline fonts in a picture box control in Visual Basic for Windows to work around the limitation in text boxes.

More Information:

The example below is one way of simulating a text box's contents in a variety of fonts.

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Place a picture box on Form1, and double-click on the picture box to open the Code window. Add the following code to the Click event. Notice that the font properties are a Boolean type (that is, -1 = True and 0 = False).

```
Sub Picture1_Click ( )

'** The word "Hello, " will be in FontBold.
        temp$ = "Hello, "
        Picture1.FontBold = -1
        Picture1.FontItalic = 0
        Picture1.FontStrikethru = 0
        Picture1.FontUnderline = 0
        Picture1.Print temp$

'** Need to program the next location to print in FontItalic.
        Picture1.Currentx = 500
        Picture1.Currenty = 0
        Picture1.FontBold = 0
        Picture1.FontItalic = -1
        Picture1.FontStrikethru = 0
        Picture1.FontUnderline = 0
        temp$ = " there!"
        Picture1.Print temp$

'** Need to program location to print in FontStrikethru.
        Picture1.Currentx = 1100
        Picture1.Currenty = 0
        Picture1.FontBold = 0
        Picture1.FontItalic = 0
        Picture1.FontUnderline = 0
        Picture1.FontStrikethru = -1
        temp$ = "This"
```

```
        Picture1.Print temp$

'** Need to program location to print in FontUnderline.
        Picture1.Currentx = 0
        Picture1.Currenty = 200
        Picture1.FontBold = 0
        Picture1.FontItalic = 0
        Picture1.FontStrikethru = 0
        Picture1.FontUnderline = -1
        temp$ = "is a test."
        Picture1.Print temp$

End Sub
```

Notice that the CurrentX and CurrentY properties are used to place the text at a certain location in the picture box. This example is rather simple, but its purpose is to give you an idea on how to simulate a text box in Visual Basic for Windows to be more flexible with a mix of the different types of fonts available.

VB Uses Bitmap Fonts when TrueType FontSize is Less than 8

Article ID: Q84483

Summary:

The Microsoft Windows version 3.1 operating environment provides you with TrueType scalable fonts that can be used in Microsoft Visual Basic for Windows applications. Visual Basic for Windows supports TrueType fonts for font sizes of 7 or greater depending on the video driver installed. Smaller fonts are mapped to available bitmap fonts, based on the fonts available for the video driver installed. There is no way to force Visual Basic for Windows to use TrueType fonts for font sizes less than 7. This is not a problem with Visual Basic for Windows, but rather a function of how Windows manages fonts.

More Information:

Microsoft Windows 3.1 utilizes automatic bitmap font substitution, which is done to preserve readability at small sizes when they are displayed. At very small point sizes (4 to 7 points on standard VGA video resolutions), most Type 2 fonts are substituted with a hand-tuned bitmap font to preserve readability. This can cause the style of the font to change. For example, the Times New Roman font shipped with Windows version 3.1 appears as the Small Fonts font for sizes 4 - 6 and MS Serif for sizes 6.25 - 8.25, rather than its native face it has at larger sizes.

The program below demonstrates this scenario. The program attempts to print a message using the Arial font in sizes from 1 to 9. Visual Basic for Windows uses the font Small Fonts for font sizes less than 7 and depending on the video driver installed may use Arial for sizes between 7 and 8.25. Using the standard VGA driver, Arial is used for fonts sizes greater then 8.25.

Steps to Demonstrate This Behavior
---

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Enter the following code into the Form_Click procedure:

```
Sub Form_Click ()
    For i = 1 To 9 Step .25
        FontName = "Arial"
        FontSize = i
        Print Str$(i); Chr$(9); Str$(FontSize); Chr$(9); FontName
    Next i
End Sub
```

3. Press the F5 key to run the program, and click anywhere on the form. Notice that the Arial TrueType font is used only for font sizes of 8.25 or larger.

Overflow Error Plotting Points Far Outside Bounds of Control

Article ID: Q81953

Summary:

Visual Basic for Windows may give an Overflow error when you plot points on a form or picture box if a point's coordinates far exceed the borders and scale of the form or control. The point at which overflow occurs depends on the ScaleMode property value and the points plotted. In the case of ScaleMode = 0 (User Defined Scale), the size of the form or picture box and the scale chosen are also determinants.

A workaround is to trap the error and use a RESUME NEXT statement to exit the error handler. The example below contains the necessary code to trap the Overflow error.

More Information:

Before Visual Basic for Windows can plot a point, it must first convert the coordinates into their absolute location in twips. If, after the conversion, one or both coordinates are greater than 32,767 or less than -32,768, an Overflow error is generated. The following chart lists the ScaleModes, their equivalence in twips, and the values that will cause a coordinate (z) to overflow:

ScaleMode	Equivalents in Twips (Tp)	Overflow Point (z)
0 (User defined)	User defined	User defined (see example)
1 (Twips)	1 twip = 1 twip	(z < -32768) or (z > 32767)
2 (Point)	1 point = 20 twips	(z < -1638) or (z > 1638)
3 (Pixel)	System dependent	System dependent
4 (Character)	x-axis=120 twips/char y-axis=240 twips/char	(x < -273) or (x > 273) (y < -136) or (y > 136)
5 (Inch)	1 Inch = 1440 twips	(z < -22) or (z > 22)
6 (Millimeter)	1 mm = 56.7 twips	(z < -577) or (z > 577)
7 (Centimeter)	1 cm = 567 twips	(z < -57) or (z > 57)

The example below can be used to determine the value that generates the Overflow error for ScaleMode 0 or 3.

Example
----

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Add the following controls to Form1:

Control	Name (use CtlName in Visual Basic 1.0 for Windows)
Text box	Text1
Command button	Command1

3. Set the MultiLine property for Text1 to True. With ScaleMode = 0

only, the overflow value is dependent upon the size of the picture box or form. If you are testing the overflow value with ScaleMode = 0, you must size the form appropriately.

4. Add the following code to the Form1 Form_Load event procedure:

```
Sub Form_Load ()
   Command1.Caption = "Find Ranges"

   '* Change ScaleMode to see different results.
   Form1.ScaleMode =  3   ' PIXEL.
End Sub
```

5. Add the following code to the Command1_Click event procedure:

```
Sub Command1_Click ()
   CR$ = Chr$(13) + Chr$(10)  ' Carriage return.

   X = FindValue("X")
   Y = FindValue("Y")

   Text1.Text = "Valid value when..."
   Text1.Text = Text1.Text + CR$ + "-" + Str$(X) + " < X < " + Str$(X)
   Text1.Text = Text1.Text + CR$ + "-" + Str$(Y) + " < Y < " + Str$(Y)
End Sub
```

6. Add the following general purpose function to the general Declarations section:

```
Function FindValue (Which$)
   On Error GoTo rlhandler

   HiValue = 100000
   LoValue = 0
   Errored = FALSE
   ' Do binary select.
   Do
      NewCheck = Value
      If Errored Then
         Value = HiValue - (HiValue - LoValue) \ 2
      Else
         Value = LoValue + (HiValue - LoValue) \ 2
      End If

      If Which$ = "X" Then
          Form1.PSet (Value, 0)
      Else
         Form1.PSet (0, Value)
      End If

      If ErrorNum = 6 Then
          HiValue = Value
         ErrorNum = 0
      Else
         LoValue = Value
      End If
   Loop Until NewCheck = Value
```

```
      FindValue = Value

      Exit Function

   rlhandler:
      ' Err = 6 is OverFlow error.
      If Err = 6 Then
         ErrorNum = Err
      Else
         Form1.Print Err
      End If
   Resume Next

End Function
```

7. In Visual Basic version 1.0 for Windows, add the following to the general declarations section of Form1:

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

8. From the Run menu, choose Start (or press the F5 key), and click on the Command1 button to calculate the point at which the X and Y coordinates generate an Overflow error.

When the above Click event is triggered, Visual Basic for Windows will try to set a point on the form. Past the border, Visual Basic for Windows is plotting points that exceed the visual scope of the control. Once the program traps the Overflow error, the text box will display the valid range of coordinates you can use that will not generate the Overflow error.

How to Make a Push Button with a Bitmap in Visual Basic

Article ID: Q78478

Summary:

Command buttons in Visual Basic for Windows are limited to a single line of text and one background color (gray). The 3D command button shipped in the Professional Editions of Visual Basic version 2.0 and 3.0 for Windows does have the capability of displaying bitmaps within a command button in Visual Basic for Windows. However, there is no way to alter the background or border colors to change its appearance. You can create the look and feel of a command button by using a picture control and manipulating the DrawMode in conjunction with the Line method. Using a picture control also allows you to display the "command button" in any color with multiple lines of caption text.

More Information:

The technique (demonstrated further below) simulates the effect of pressing a command button by using the Line method with the BF option (Box Fill) in invert mode each time a MouseUp or MouseDown event occurs for the picture control. To add multiline text to the "button," either print to the picture box or add the text permanently to the bitmap.

The steps to create a customized "command button" are as follows:

1. Start Visual Basic for Windows, or choose New Project from the File menu (press ALT, F, N) if Visual Basic for Windows is already running. Form1 will be created by default.

2. Put a picture control (Picture1) on Form1.

3. Set the properties for Picture1 as given in the chart below:

Property	Value
AutoRedraw	True
AutoSize	True
BorderStyle	0-None
DrawMode	6-Invert

4. Assign the Picture property of Picture1 to the bitmap of your choice. For example, choose ARW01DN.ICO from the ARROWS subdirectory of the ICONS directory shipped with Visual Basic for Windows. This is a good example of a bitmap with a three dimensional appearance.

5. Enter the following code in the Picture1_DblClick event procedure of Picture1:

   ```
   Sub Picture1_DblClick ()
       Picture1.Line (0, 0)-(Picture1.width, Picture1.height), , BF
   End Sub
   ```

   Note: This code is necessary to avoid getting the bitmap stuck in an inverted state because of Mouse messages being processed out of

order or from piling up due to fast clicking.

6. Enter the following code in the Picture1_MouseDown event procedure of Picture1:

```
Sub Picture1_MouseDown (Button As Integer, Shift As Integer, X As
                        Single, Y As Single)  ' Append to above line
    Picture1.Line (0, 0)-(Picture1.width, Picture1.height), , BF
End Sub
```

7. Enter the following code in the Picture1_MouseUp event procedure of Picture1:

```
Sub Picture1_MouseUp (Button As Integer, Shift As Integer,
                      X As Single, Y As Single) ' Append to above line.
    Picture1.Line (0, 0)-(Picture1.width, Picture1.height), , BF
End Sub
```

8. Add the following code to the Picture1_KeyUp event procedure for Picture1:

```
Sub Picture1_KeyUp (KeyCode As Integer, Shift As Integer)
    '* Check to see if the ENTER key was pressed.  If so, restore
    '* the picture image.
    If KeyCode = 13 Then
      Picture1.Line (0, 0)-(Picture1.width, Picture1.height), , BF
    End If
End Sub
```

9. Add the following code to the Picture1_KeyDown event procedure for Picture1:

```
Sub Picture1_KeyDown (KeyCode As Integer, Shift As Integer)
    '* Check to see if the ENTER key was pressed.  If so, invert
    '* the picture image.
    If KeyCode = 13 Then
      Picture1.Line (0, 0)-(Picture1.width, Picture1.height), , BF
    End If
End Sub
```

10. From the Run menu, choose Start. Click on the picture box. The image of the picture should be inverted while the mouse button is down, giving the visual effect of a button press.

How to Use FillPolygonRgn API to Fill Shape in Visual Basic

Article ID: Q81470

Summary:

Microsoft Visual Basic versions 2.0 and later for Windows include the Shape control which can be used for creating and filling six different geometric shapes. Alternatively, you can create a polygon region on a form or picture and fill it with a color, using the CreatePolygonRgn and FillRgn Windows API calls to draw and fill areas of the screen with color. Geometric shapes not provided with the Shape control, such as a triangle, can be created using this method.

More Information:

To draw a polygon on a form or picture control, you can use the Polygon API call; this will draw the edge of the polygon. You can then use CreatePolygonRgn to create an area that you can paint and use FillRgn to fill it with a color. Using these Windows API calls allows you to pick the points, the number of points, and to choose the color or brush to fill with.

The API calls used in the following example should be declared in the general Declarations section of your form. They are as follows:

API Call	Description
CreatePolygonRgn	Creates a polygonal region
GetStockObject	Retrieves a handle to one of the predefined stock pens, brushes, or fonts
FillRgn	Fills the region specified by the hRgn parameter with the brush specified by the hBrush parameter
Polygon	Draws a polygon consisting of two or more points connected by lines

Code Example

The following code example shows how to create a black triangle on a form. To change the program to create other shapes, add points to the array.

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. From the File menu, choose New Module (press ALT, F, M). Module1 is created by default.

3. Add the following code to the general declarations section of Module1 (in Visual Basic version 1.0 for Windows, add to Global.Bas):

```
Type Coord      ' This is the type structure for the x and y
  x As Integer  ' coordinates for the polygonal region.
```

```
  y As Integer
End Type

' All of the following Declare statements must appear on one line.
Declare Function CreatePolygonRgn Lib "gdi"
          (lpPoints As Any, ByVal nCount As Integer,
           ByVal nPolyFillMode As Integer) As Integer

Declare Function Polygon Lib "gdi"
          (ByVal hDC As Integer, lpPoints As Any,
           ByVal nCount As Integer) As Integer

Declare Function FillRgn Lib "gdi"
          (ByVal hDC As Integer, ByVal hRgn As Integer,
           ByVal hBrush As Integer) As Integer

Declare Function GetStockObject Lib "gdi"
          (ByVal nIndex As Integer) As Integer

Global Const ALTERNATE = 1 ' ALTERNATE and WINDING are
Global Const WINDING = 2   ' constants for FillMode.
Global Const BLACKBRUSH = 4' Constant for brush type.
```

2. Add the following code to the Form_Click event for Form1:

```
Sub Form_Click ()
  ' Dimension coordinate array.
  ReDim poly(1 To 3) As Coord
  ' Number of vertices in polygon.
  NumCoords% = 3
  ' Set scalemode to pixels to set up points of triangle.
  form1.scalemode = 3
  ' Assign values to points.
  poly(1).x = form1.scalewidth / 2
  poly(1).y = form1.scaleheight / 2
  poly(2).x = form1.scalewidth / 4
  poly(2).y = 3 * form1.scaleheight / 4
  poly(3).x = 3 * form1.scalewidth / 4
  poly(3).y = 3 * form1.scaleheight / 4
  ' Sets background color to red for contrast.
  form1.backcolor = &HFF
  ' Polygon function creates unfilled polygon on screen.
  ' Remark FillRgn statement to see results.
  bool% = Polygon(form1.hdc, poly(1), NumCoords%)
  ' Gets stock black brush.
  hbrush% = GetStockObject(BLACKBRUSH)
  ' Creates region to fill with color.
  hrgn% = CreatePolygonRgn(poly(1), NumCoords%, ALTERNATE)
  ' If the creation of the region was successful then color.
  If hrgn% Then bool% = FillRgn(form1.hdc, hrgn%, hbrush%)
  ' Print out some information.
  Print "FillRgn Return : ";bool%
  Print "HRgn : "; hrgn%
  Print "Hbrush : "; hbrush%

End Sub
```

3. Run the program.

You should initially see a blank form. Click the form; a red background with a black triangle on it should be displayed. You can try different numbers of vertices by adding elements to the poly array and updating NumCoords. Different colors and brushes can be substituted as desired.

Note: If you try to fill a region with coordinates beyond the visible form, the CreatePolygonRgn function call will return a zero, meaning it was unsuccessful. The FillRgn will not work if the CreatePolygonRgn function was unsuccessful. All you will see is the outline created by the Polygon function. You should make certain that the vertices are all within the viewable form.

How to Send an HBITMAP to Windows API Function Calls from VB

Article ID: Q71260

Summary:

Several Windows API functions require the HBITMAP data type. Visual Basic for Windows does not have a HBITMAP data type. This article explains how to send the equivalent Visual Basic for Windows HBITMAP handle of a picture control to a Windows API function call.

More Information:

The HBITMAP data type represents a 16-bit index to GDIs physical drawing object. Several Windows API routines need the HBITMAP data type as an argument. Sending the [picture-control].Picture as an argument is the equivalent in Visual Basic for Windows.

The code sample below demonstrates how to send HBITMAP to the Windows API function ModifyMenu:

```
Declare Function SetMenuItemBitMaps% Lib "user" (ByVal hMenu%,
                                                 ByVal  nPos%,
                                                 ByVal wFlag%,
                                                 ByVal BitmapUnChecked%,
                                                 ByVal hBitmapChecked%)
```

Note: The above Declare statement must be written on just one line.

The SetMenuItemBitMap takes five arguments. The fourth and fifth arguments are HBITMAP data types.

The following code segment will associate the specified bitmap Picture1.Picture in place of the default check mark:

```
X% = SetMenuItemBitMap(hMenu%, menuID%,0,0, Picture1.Picture)
```

How to Rotate a Bitmap in VB for Windows
Article ID: Q80406
Summary:

This article contains a program example that uses Visual Basic for Windows statements and functions to rotate a bitmap.

More Information:

Steps to Create Example Program
---

1. Run Visual Basic for Windows, or from the File menu, choose New Project (ALT, F, N) if Visual Basic for Windows is already running. Form1 will be created by default.

2. Place two picture boxes named Picture1 and Picture2 on Form1.

3. Set the ScaleMode property of both picture boxes to 3 - Pixel.

4. Set the AutoSize property of Picture1 to True (-1).

5. Set the AutoRedraw property of Picture1 and Picture2 to True (-1).

6. Place a command button named Command1 on Form1.

7. Enter the following code in the Command1_Click event procedure:

```
' Example of how to call bmp_rotate.
Sub Command1_Click ()
    Const Pi = 3.14159265359

    For angle = Pi / 6 To 2 * Pi Step Pi / 6
       picture2.Cls
       Call bmp_rotate(picture1, picture2, angle)
    Next
End Sub
```

8. Enter the following code in the general Declarations section:

```
' bmp_rotate(pic1, pic2, theta)
' Rotate the image in a picture box.
'   pic1 is the picture box with the bitmap to rotate
'   pic2 is the picture box to receive the rotated bitmap
'   theta is the angle of rotation
'
Sub bmp_rotate (pic1 As Control, pic2 As Control, theta!)
  Const Pi = 3.14159265359
  Dim c1x As Integer  ' Center of pic1.
  Dim c1y As Integer  '   "
  Dim c2x As Integer  ' Center of pic2.
  Dim c2y As Integer  '   "
  Dim a As Single     ' Angle of c2 to p2.
  Dim r As Integer    ' Radius from c2 to p2.
  Dim p1x As Integer  ' Position on pic1.
  Dim p1y As Integer  '   "
  Dim p2x As Integer  ' Position on pic2.
  Dim p2y As Integer  '   "
```

```
    Dim n As Integer    ' Max width or height of pic2.

    ' Compute the centers.
    c1x = pic1.scalewidth / 2
    c1y = pic1.scaleheight / 2
    c2x = pic2.scalewidth / 2
    c2y = pic2.scaleheight / 2

    ' Compute the image size.
    n = pic2.scalewidth
    If n < pic2.scaleheight Then n = pic2.scaleheight
    n = n / 2 - 1
    ' For each pixel position on pic2.
    For p2x = 0 To n
       For p2y = 0 To n
          ' Compute polar coordinate of p2.
          If p2x = 0 Then
            a = Pi / 2
          Else
            a = Atn(p2y / p2x)
          End If
          r = Sqr(1& * p2x * p2x + 1& * p2y * p2y)

          ' Compute rotated position of p1.
          p1x = r * Cos(a + theta)
          p1y = r * Sin(a + theta)

          ' Copy pixels, 4 quadrants at once.
          c0& = pic1.Point(c1x + p1x, c1y + p1y)
          c1& = pic1.Point(c1x - p1x, c1y - p1y)
          c2& = pic1.Point(c1x + p1y, c1y - p1x)
          c3& = pic1.Point(c1x - p1y, c1y + p1x)
          If c0& <> -1 Then pic2.PSet (c2x + p2x, c2y + p2y),c0&
          If c1& <> -1 Then pic2.PSet (c2x - p2x, c2y - p2y),c1&
          If c2& <> -1 Then pic2.PSet (c2x + p2y, c2y - p2x),c2&
          If c3& <> -1 Then pic2.PSet (c2x - p2y, c2y + p2x),c3&
       Next
       ' Allow pending Windows messages to be processed.
       t% = DoEvents()
    Next
 End Sub
```

 9. Assign a bitmap image to the Picture1 Picture property.

10. To start the program, press F5, then click on Command1. The program rotates the image of Picture1 by 30 degrees and places the rotated image in Picture2. It continues to draw the image rotated at successive multiples of 30 degrees until it has rotated the picture by 360 degrees.

To save the new bitmap created in Picture2, you can use the following statement:

```
SavePicture Picture2.Image, "filename.bmp"
```

How to Create a Transparent Bitmap Using Visual Basic

Article ID: Q94961

Summary:

A transparent image shows the background behind it instead of the image itself. You can use an icon editor such as the IconWorks sample program provided with Visual Basic to create icons that contain transparent parts. This article shows you how to make certain parts of a bitmap transparent.

More Information:

Here are the six general steps required to create a transparent bitmap:

1. Store the area, or background, where the bitmap is going to be drawn.
2. Create a monochrome mask of the bitmap that identifies the transparent areas of the bitmap by using a white pixel to indicate transparent areas and a black pixel to indicate non-transparent areas of the bitmap.
3. Combine the pixels of the monochrome mask with the background bitmap using the And binary operator. The area of the background where the non-transparent portion of the bitmap will appear is made black.
4. Combine an inverted copy of the monochrome mask (step 2) with the source bitmap using the And binary operator. The transparent areas of the source bitmap will be made black.
5. Combine the modified background (step 3) with the modified source bitmap (step 4) using the Xor binary operator. The background will show through the transparent portions of the bitmap.
6. Copy the resulting bitmap to the background

Example Code
---

1. Run Visual Basic, or from the File menu, choose New Project (ALT, F, N) if Visual Basic is already running. Form1 is created by default.
2. Add the following controls to Form1 with the associated property values:

Control	Name (or CtlName)	Property Settings
Picture	pictSource	Picture ="WINDOWS\THATCH.BMP"
Picture	pictDest	Picture ="WINDOWS\ARCHES.BMP"
Command button	cmdCopy	Caption ="Copy"

3. From the File menu, choose New Module (ALT, F, M). Module1 is created.
4. Add the following code to the cmdCopy_Click event procedure of Form1. This code calls the TransparentBlt() function to copy a source bitmap to a destination (background) picture control. White (QBColor(15)) areas of the bitmap are made transparent against the background bitmap.

```
Sub cmdCopy_Click ()
   Call TransparentBlt(pictDest, pictSource.Picture, 10, 10, QBColor(15))
End Sub
```

5. Add the following code the general declarations section of Module1. Enter each Declare as a single line:

```
Type bitmap
   bmType As Integer
   bmWidth As Integer
   bmHeight As Integer
```

```
    bmWidthBytes As Integer
    bmPlanes As String * 1
    bmBitsPixel As String * 1
    bmBits As Long
End Type
Declare Function BitBlt Lib "GDI" (ByVal srchDC As Integer, ByVal srcX
    As Integer, ByVal srcY As Integer, ByVal srcW As Integer, ByVal srcH
    As Integer, ByVal desthDC As Integer, ByVal destX As Integer, ByVal
    destY As Integer, ByVal op As Long) As Integer
Declare Function SetBkColor Lib "GDI" (ByVal hDC As Integer, ByVal
    cColor As Long) As Long
Declare Function CreateCompatibleDC Lib "GDI" (ByVal hDC As Integer)
    As Integer
Declare Function DeleteDC Lib "GDI" (ByVal hDC As Integer) As Integer
Declare Function CreateBitmap Lib "GDI" (ByVal nWidth As Integer, ByVal
    nHeight As Integer, ByVal cbPlanes As Integer, ByVal cbBits As
    Integer, lpvBits As Any) As Integer
Declare Function CreateCompatibleBitmap Lib "GDI" (ByVal hDC As Integer,
    ByVal nWidth As Integer, ByVal nHeight As Integer) As Integer
Declare Function SelectObject Lib "GDI" (ByVal hDC As Integer, ByVal
    hObject As Integer) As Integer
Declare Function DeleteObject Lib "GDI" (ByVal hObject As Integer) As
    Integer
Declare Function GetObject Lib "GDI" (ByVal hObject As Integer, ByVal
    nCount As Integer, bmp As Any) As Integer
Const SRCCOPY = &HCC0020
Const SRCAND = &H8800C6
Const SRCPAINT = &HEE0086
Const NOTSRCCOPY = &H330008
```

6. Add the following Sub procedure to the general declarations section of Module1. TransparentBlt() accepts six parameters: a destination picture control (dest), a source bitmap to become transparent (srcBmp), the X,Y coordinates in pixels where you want to place the source bitmap on the destination (destX and destY), and the RGB value for the color you want to be transparent. TransparentBlt() copies the source bitmap to any X,Y location on the background making areas transparent.

```
Sub TransparentBlt (dest As Control, ByVal srcBmp As Integer, ByVal
    destX As Integer, ByVal destY As Integer, ByVal TransColor As Long)
    Const PIXEL = 3
    Dim destScale As Integer
    Dim srcDC As Integer  'source bitmap (color)
    Dim saveDC As Integer 'backup copy of source bitmap
    Dim maskDC As Integer 'mask bitmap (monochrome)
    Dim invDC As Integer  'inverse of mask bitmap (monochrome)
    Dim resultDC As Integer 'combination of source bitmap & background
    Dim bmp As bitmap 'description of the source bitmap
    Dim hResultBmp As Integer 'Bitmap combination of source & background
    Dim hSaveBmp As Integer 'Bitmap stores backup copy of source bitmap
    Dim hMaskBmp As Integer 'Bitmap stores mask (monochrome)
    Dim hInvBmp As Integer  'Bitmap holds inverse of mask (monochrome)
    Dim hPrevBmp As Integer 'Bitmap holds previous bitmap selected in DC
    Dim hSrcPrevBmp As Integer  'Holds previous bitmap in source DC
    Dim hSavePrevBmp As Integer 'Holds previous bitmap in saved DC
    Dim hDestPrevBmp As Integer 'Holds previous bitmap in destination DC
    Dim hMaskPrevBmp As Integer 'Holds previous bitmap in the mask DC
```

```
Dim hInvPrevBmp As Integer  'Holds previous bitmap in inverted mask DC
Dim OrigColor As Long  'Holds original background color from source DC
Dim Success As Integer 'Stores result of call to Windows API
If TypeOf dest Is PictureBox Then 'Ensure objects are picture boxes
  destScale = dest.ScaleMode 'Store ScaleMode to restore later
  dest.ScaleMode = PIXEL 'Set ScaleMode to pixels for Windows GDI
  'Retrieve bitmap to get width (bmp.bmWidth) & height (bmp.bmHeight)
  Success = GetObject(srcBmp, Len(bmp), bmp)
  srcDC = CreateCompatibleDC(dest.hDC)    'Create DC to hold stage
  saveDC = CreateCompatibleDC(dest.hDC)   'Create DC to hold stage
  maskDC = CreateCompatibleDC(dest.hDC)   'Create DC to hold stage
  invDC = CreateCompatibleDC(dest.hDC)    'Create DC to hold stage
  resultDC = CreateCompatibleDC(dest.hDC) 'Create DC to hold stage
  'Create monochrome bitmaps for the mask-related bitmaps:
  hMaskBmp = CreateBitmap(bmp.bmWidth, bmp.bmHeight, 1, 1, ByVal 0&)
  hInvBmp = CreateBitmap(bmp.bmWidth, bmp.bmHeight, 1, 1, ByVal 0&)
  'Create color bitmaps for final result & stored copy of source
  hResultBmp = CreateCompatibleBitmap(dest.hDC, bmp.bmWidth,
     bmp.bmHeight)
  hSaveBmp = CreateCompatibleBitmap(dest.hDC, bmp.bmWidth,
     bmp.bmHeight)
  hSrcPrevBmp = SelectObject(srcDC, srcBmp)     'Select bitmap in DC
  hSavePrevBmp = SelectObject(saveDC, hSaveBmp) 'Select bitmap in DC
  hMaskPrevBmp = SelectObject(maskDC, hMaskBmp) 'Select bitmap in DC
  hInvPrevBmp = SelectObject(invDC, hInvBmp)    'Select bitmap in DC
  hDestPrevBmp = SelectObject(resultDC, hResultBmp) 'Select bitmap
  Success = BitBlt(saveDC, 0, 0, bmp.bmWidth, bmp.bmHeight, srcDC,
     0, 0, SRCCOPY) 'Make backup of source bitmap to restore later
  'Create mask: set background color of source to transparent color.
  OrigColor = SetBkColor(srcDC, TransColor)
  Success = BitBlt(maskDC, 0, 0, bmp.bmWidth, bmp.bmHeight, srcDC,
     0, 0, SRCCOPY)
  TransColor = SetBkColor(srcDC, OrigColor)
  'Create inverse of mask to AND w/ source & combine w/ background.
  Success = BitBlt(invDC, 0, 0, bmp.bmWidth, bmp.bmHeight, maskDC,
    0, 0, NOTSRCCOPY)
  'Copy background bitmap to result & create final transparent bitmap
  Success = BitBlt(resultDC, 0, 0, bmp.bmWidth, bmp.bmHeight,
     dest.hDC, destX, destY, SRCCOPY)
  'AND mask bitmap w/ result DC to punch hole in the background by
  'painting black area for non-transparent portion of source bitmap.
  Success = BitBlt(resultDC, 0, 0, bmp.bmWidth, bmp.bmHeight,
     maskDC, 0, 0, SRCAND)
  'AND inverse mask w/ source bitmap to turn off bits associated
  'with transparent area of source bitmap by making it black.
  Success = BitBlt(srcDC, 0, 0, bmp.bmWidth, bmp.bmHeight, invDC,
     0, 0, SRCAND)
  'XOR result w/ source bitmap to make background show through.
  Success = BitBlt(resultDC, 0, 0, bmp.bmWidth, bmp.bmHeight,
     srcDC, 0, 0, SRCPAINT)
  Success = BitBlt(dest.hDC, destX, destY, bmp.bmWidth, bmp.bmHeight,
     resultDC, 0, 0, SRCCOPY) 'Display transparent bitmap on backgrnd
  Success = BitBlt(srcDC, 0, 0, bmp.bmWidth, bmp.bmHeight, saveDC,
     0, 0, SRCCOPY) 'Restore backup of bitmap.
  hPrevBmp = SelectObject(srcDC, hSrcPrevBmp) 'Select orig object
  hPrevBmp = SelectObject(saveDC, hSavePrevBmp) 'Select orig object
  hPrevBmp = SelectObject(resultDC, hDestPrevBmp) 'Select orig object
```

```
      hPrevBmp = SelectObject(maskDC, hMaskPrevBmp) 'Select orig object
      hPrevBmp = SelectObject(invDC, hInvPrevBmp) 'Select orig object
      Success = DeleteObject(hSaveBmp)   'Deallocate system resources.
      Success = DeleteObject(hMaskBmp)   'Deallocate system resources.
      Success = DeleteObject(hInvBmp)    'Deallocate system resources.
      Success = DeleteObject(hResultBmp) 'Deallocate system resources.
      Success = DeleteDC(srcDC)          'Deallocate system resources.
      Success = DeleteDC(saveDC)         'Deallocate system resources.
      Success = DeleteDC(invDC)          'Deallocate system resources.
      Success = DeleteDC(maskDC)         'Deallocate system resources.
      Success = DeleteDC(resultDC)       'Deallocate system resources.
      dest.ScaleMode = destScale 'Restore ScaleMode of destination.
   End If
End Sub
```

7. From the Run menu, choose Start (ALT, R, S) to run the program.
8. Click the Copy button. The thatched pattern in the first picture is copied onto the second picture (an image of arches) making the arches show through areas of the previously white thatched pattern.

How to Copy Entire Screen into a Picture Box in VB for Windows

Article ID: Q80670

Summary:

Using the Windows API call BitBlt, you can capture the entire Microsoft Windows screen and place the image into a Microsoft Visual Basic for Windows picture box. You first get the handle to the desktop, then use the desktop window handle to get the handle to the desktop's device context (hDC), and finally use the Windows API call BitBlt to copy the screen into the Picture property of a Visual Basic for Windows picture box control.

More Information:

Example
----

1. Start Visual Basic for Windows (VB.EXE). Form1 is created by default.

2. Create a picture box (Picture1) on Form1.

3. Set the following properties:

Control	Property	Value
Picture1	AutoRedraw	True
Picture1	Visible	False

4. Add the following code:

Global.Bas

```
Type lrect
    left As Integer
    top As Integer
    right As Integer
    bottom As Integer
End Type
Declare Function GetDesktopWindow Lib "user" () As Integer
Declare Function GetDC Lib "user" (ByVal hWnd%) As Integer

' Note: The following Declare should be on one line:
Declare Function BitBlt Lib "GDI" (ByVal hDestDC%,
                                   ByVal X%,
                                   ByVal Y%,
                                   ByVal nWidth%,
                                   ByVal nHeight%,
                                   ByVal hSrcDC%,
                                   ByVal XSrc%,
                                   ByVal YSrc%,
                                   ByVal dwRop&
                                  ) As Integer

' Note: The following Declare should be on one line:
Declare Function ReleaseDC Lib "User"(ByVal hWnd As Integer,
```

```
                                        ByVal hDC As Integer
                                       ) As Integer

Declare Sub GetWindowRect Lib "User" (ByVal hWnd%, lpRect As lrect)
Global Const True = -1
Global Const False = 0
Global TwipsPerPixel As Single

Form1
-----

Sub Form_Click ()
  Call GrabScreen
End Sub

Sub GrabScreen ()

    Dim winSize As lrect

    ' Assign information of the source bitmap.
    ' Note that BitBlt requires coordinates in pixels.
    hwndSrc% = GetDesktopWindow()
    hSrcDC% = GetDC(hwndSrc%)
    XSrc% = 0: YSrc% = 0
    Call GetWindowRect(hwndSrc%, winSize)
    nWidth% = winSize.right             ' Units in pixels.
    nHeight% = winSize.bottom           ' Units in pixels.

    ' Assign informate of the destination bitmap.
    hDestDC% = Form1.Picture1.hDC
    x% = 0: Y% = 0

    ' Set global variable TwipsPerPixel and use to set
    ' picture box to same size as screen being grabbed.
    ' If picture box not the same size as picture being
    ' BitBlt'ed to it, it will chop off all that does not
    ' fit in the picture box.
    GetTwipsPerPixel
    Form1.Picture1.Top = 0
    Form1.Picture1.Left = 0
    Form1.Picture1.Width = (nWidth% + 1) * TwipsPerPixel
    Form1.Picture1.Height = (nHeight% + 1) * TwipsPerPixel

    ' Assign the value of the constant SRCOPYY to the Raster operation.
    dwRop& = &HCC0020

    ' Note function call must be on one line:
    Suc% = BitBlt(hDestDC%, x%, Y%, nWidth%, nHeight%,
                  hSrcDC%, XSrc%, YSrc%, dwRop&)

    ' Release the DeskTopWindow's hDC to Windows.
    ' Windows may hang if this is not done.
    Dmy% = ReleaseDC(hwndSrc%, hSrcDC%)

    'Make the picture box visible.
    Form1.Picture1.Visible = True
End Sub
```

```
Sub GetTwipsPerPixel ()
    ' Set a global variable with the Twips to Pixel ratio.
    Form1.ScaleMode = 3
    NumPix = Form1.ScaleHeight
    Form1.ScaleMode = 1
    TwipsPerPixel = Form1.ScaleHeight / NumPix
End Sub
```

5. Run the program and click on the form.

6. With the mouse, change the size of the form to see more of the picture box. With a little work, you can use this as a "screen saver" program.

How to Flood Fill (Paint) in VB using ExtFloodFill Windows API

Article ID: Q71103

Summary:

You can fill an area on a window in Visual Basic through a Windows API function call. Depending on the type of fill to be performed, you can use the ExtFloodFill function to achieve the desired effect. This feature is similar to the paint feature found in painting programs.

This information applies to Microsoft Visual Basic programming system version 1.0 for Windows.

More Information:

The Windows API ExtFloodFill function call fills an area of the display surface with the current brush, as shown in the example below.

Code Example

From the VB.EXE Code menu, choose View Code, and enter the following code (on just one line) for Form1 (using [general] from the Object box and [declarations] from the Procedure box):

```
Declare Function ExtFloodFill Lib "GDI" (ByVal hdc%, ByVal i%,
   ByVal i%, ByVal w&, ByVal i%) As Integer
```

To demonstrate several fill examples, create a picture box called Picture1. Set the following properties:

```
AutoSize = TRUE           ' Scale picture to size of imported picture.
FillColor = &HFF00FF      ' This will be the selected fill color.
FillStyle = Solid         ' Necessary to create a fill pattern.
Picture = Chess.bmp       ' This should be in your Windows directory.
```

Create a push button in a location that will not be overlapped by Picture1. Within the Click event, create the following code:

```
Sub Command1_Click ()
 ' Make sure that the FillStyle is not transparent.
 ' crColor& specifies the color for the boundary.
  Const FLOODFILLBORDER = 0  ' Fill until crColor& color encountered.
  Const FLOODFILLSURFACE = 1 ' Fill surface until crColor& color not
                             ' encountered.
  X% = 1
  Y% = 1
  crColor& = RGB(0, 0, 0)
  wFillType% = FLOODFILLSURFACE
  Suc% = ExtFloodFill(picture1.hDC, X%, Y%, crColor&, wFillType%)
End Sub
```

When you click on the push button, the black background will change to the FillColor. The fill area is defined by the color specified by crColor&. Filling continues outward from (X%,Y%) as long as the color is encountered.

Now change the related code to represent the following:

```
   crColor& = RGB(255, 0, 0)  'Color to look for.
   wFillType% = FLOODFILLBORDER
   Suc% =  ExtFloodFill(picture1.hDC, X%, Y%, crColor&, wFillType%)
```

Executing the push button will now fill the area until crColor& is encountered. In the first example, the fill was performed while the color was encountered; in the second example, the fill was performed while the color was NOT encountered. In the last example, everything is changed except the "floating pawn".

Reference(s):

"Programming Windows: the Microsoft Guide to Writing Applications for Windows 3," by Charles Petzold, Microsoft Press, 1990

"Microsoft Windows Software Development Kit: Reference Volume 1," version 3.0

WINSDK.HLP file shipped with Microsoft Windows 3.0 Software Development Kit

VB CDK: Example of Subclassing a Visual Basic Form

Article ID: Q83806

Summary:

The subclass procedure is a message filter that performs non-default processing for a few key messages, and passes other messages to a control's default window procedure using CallWindowProc. The CallWindowProc function passes a message to Windows, which in turn sends the message to the target window procedure. The target window procedure cannot be called directly by the subclass procedure because the target procedure is exported.

More Information:

The following code example demonstrates how to subclass a form using the Microsoft Visual Basic for Windows Custom Control Development Kit (CDK).

This example is developed using the CIRCLE.C source file from the CIRCLE1 project supplied with the CDK package. Only the file(s) that have changed from this project are included, and it is assumed that you have the additional CDK files.

```
  //=================== CIRCLE1 ==============================
  // CIRCLE.C
  // An example of subclassing a Visual Basic for Windows Form
  //==========================================================

#define  NOCOMM
#include <windows.h>

#include <vbapi.h>
#include "circle.h"

// Declare the subclass procedure.
LONG FAR PASCAL _export SbClsProc(HWND,USHORT,USHORT,LONG);

// Far pointer to the default procedure.
FARPROC lpfnOldProc = (FARPROC) NULL ;

// Get the controls parent handle(form1).
HWND    hParent ;

//----------------------------------------------------------
// Circle Control Procedure
//----------------------------------------------------------
LONG FAR PASCAL _export CircleCtlProc (HCTL hctl, HWND hwnd,
     USHORT msg, USHORT wp, LONG lp)
{
   LONG lResult ;
   switch (msg)
   {
      case WM_CREATE:
         switch (VBGetMode())
         {
            // This will only be processed during run mode.
            case MODE_RUN:
```

```
            {
               hParent = GetParent (hwnd) ;
               // Get the address instance to normal proc.
               lpfnOldProc = (FARPROC) GetWindowLong
                             (hParent, GWL_WNDPROC) ;
               // Reset the address instance to the new proc.
               SetWindowLong (hParent,
                        GWL_WNDPROC, (LONG) SbClsProc) ;
            }
            break ;
         }
         break ;
   }
   // Call the default VB for Windows proc.
   lResult = VBDefControlProc(hctl, hwnd, msg, wp, lp);
   return lResult;

}

LONG FAR PASCAL _export SbClsProc (HWND hwnd, USHORT msg,
     USHORT wp, LONG lp)
{
   switch (msg)
   {
      case WM_SIZE:
      {
      // Place size event here for example...
      }
      break;
      case WM_DESTROY:
         SetWindowLong (hwnd, GWL_WNDPROC,
                        (LONG) lpfnOldProc) ;
      break ;
   }
   // Call CircleCtlProc to process any other messages.
   return (CallWindowProc(lpfnOldProc, hwnd, msg, wp, lp));
}

;============================================================
;Circle.def - module definition file for CIRCLE3.VBX control
;============================================================
LIBRARY         CIRCLE
EXETYPE         WINDOWS
DESCRIPTION     'Visual Basic Circle Custom Control'

CODE            MOVEABLE
DATA            MOVEABLE SINGLE

HEAPSIZE        1024

EXPORTS
           WEP       @1 RESIDENTNAME
           SbClsProc @2
;------------------------------------------------------
```

Declare Currency Type to Be Double When Returning from DLL

Article ID: Q72274

Summary:

When using Microsoft Visual Basic for Windows, if you want to pass a parameter to a dynamic link library (DLL) routine, or receive a function return value of type Currency from a DLL routine written in Microsoft C, the parameter or function returned should be declared as a "double" in the C routine.

Note that C does not support the Basic Currency data type, and although specifying the parameter as type "double" in C will allow it to be passed correctly, you will have to write your own C routines to manipulate the data in the Currency variable.

More Information:

When creating a DLL function that either receives or returns a Currency data type, it may be useful to include the following declaration:

```
typedef double currency;
```

Based on this typedef, a sample DLL routine to return a currency value might be declared as follows:

```
currency FAR pascal foo(...);
```

How to Pass One-Byte Parameters from VB to DLL Routines

Article ID: Q71106

Summary:

Calling some routines in dynamic link libraries (DLLs) requires BYTE parameters in the argument list. Visual Basic for Windows possesses no BYTE data type as defined in other languages such as C, which can create DLLs. To pass a BYTE value correctly to an external FUNCTION (in a DLL), which requires a BYTE data type, you must pass an integer data type for the BYTE parameter.

More Information:

Visual BASIC for Windows has the ability to call external code in the form of dynamic link libraries (DLLs). Some of these libraries require BYTE parameters in the argument list. An example of this is located in the KEYBOARD.DRV FUNCTION as defined below:

```
    FUNCTION GetTempFileName (BYTE  cDrive,
                              LPSTR lpPrefix,
                              WORD  wUnique,
                              LPSTR lpTempFileName)
```

GetTempFileName is documented on page 4-217 of the "Microsoft Windows 3.0 Software Development Kit, Reference - Volume 1." In Visual Basic for Windows, declare the FUNCTION on one line in the main module of your code:

```
   DECLARE FUNCTION GetTempFileName LIB "keyboard.drv"
                   (BYVAL A%,  BYVAL B$, BYVAL C%, BYVAL D$)
```

Because the architecture of the 80x86 stack is segmented into word boundaries, the smallest type pushed onto the stack will be a word. Therefore, both the BYTE and the integer will be pushed onto the stack in the same manner, and require the same amount of memory. This is the reason you can use an integer data type for a BYTE data type in these types of procedure calls.

VB "Cannot Find DLL, Insert in Drive A" Using Shell

Article ID: Q80404

Summary:

When a Visual Basic for Windows application shells to a Microsoft Windows application that expects to find a dynamic link library (DLL) in its own directory, Visual Basic for Windows may generate the following error message and fail to start the application:

   Cannot Find <DLL NAME>, Please Insert in Drive A

This error occurs because the application being shelled to expects to find the DLL in the current directory, the MS-DOS path, or the Windows directory. Shelling to an application in code does not change the current directory, even if you specify the path to the application in the Shell statement.

One solution is to use Visual Basic for Windows' ChDir statement to change the current directory to the directory containing the DLL before attempting to shell to the application. An alternative solution is to copy the DLL to the Windows directory, or include the path where the DLL is located in the MS-DOS path.

More Information:

The following is a pseudocode example that shows how to use the ChDir statement to make the application's directory the current directory. The C:\APPS directory and the .EXE name MYAPP.EXE are arbitrary names selected to represent the location of the application being shelled to and an .EXE name, respectively.

```
    Sub Form_Click ()
        ChDir "c:\Apps"     ' The name of the directory containing
                            ' the needed DLL.
        x% = Shell("c:\Apps\MyApp.EXE", 1)
    End Sub
```

Note: If the application is on a different drive, use the ChDrive statement first to change drives before using the ChDir statement.

VB "Bad DLL Calling Convention" Means Stack Frame Mismatch

Article ID: Q85108

Summary:

When you call a dynamic link library (DLL) function from Visual Basic for Windows, the "Bad DLL Calling Convention" error is often caused by incorrectly omitting or including the ByVal keyword from the Declare statement or the Call statement. The ByVal keyword affects the size of data placed on the stack. Visual Basic for Windows checks the change in the position of the stack pointer to detect this error.

When Visual Basic for Windows generates the run time error "Bad DLL Calling Convention," the most common cause when calling API functions is omitting the ByVal keyword from the Declaration of the external function or from the call itself. It can also occur due to including the ByVal keyword when the function is expecting a 4 byte pointer to the parameter instead of the value itself. This changes the size (number of bytes) of the values placed on the stack, and upon return from the DLL, Visual Basic for Windows detects the change in the position of the stack frame and generates the error.

More Information:

There are two calling conventions, or inter-language protocols: the Pascal/Basic/FORTRAN calling convention, and the C calling convention. Visual Basic for Windows uses the Pascal calling convention, as do the Microsoft Window API functions and other Microsoft Basic language products. Under the Pascal convention, it is the responsibility of the called procedure to adjust or clean the stack. (In addition, parameters are pushed onto the stack in order from the leftmost parameter to the rightmost.) Because the DLL function is responsible for adjusting the stack based on the type and number of parameters it expects, Visual Basic for Windows checks the position of the stack pointer upon return from the function. If the called routine has adjusted the stack to an unexpected position, then Visual Basic for Windows generates a "Bad DLL Calling Convention" error. Visual Basic for Windows assumes a stack position discrepancy because the DLL function uses the C calling convention. With the C calling convention, the calling program is responsible for adjusting the stack immediately after the called routine returns control.

Steps to Reproduce Behavior

Create a simple DLL using Microsoft Quick C for Windows or any compiler capable of creating Windows DLLs. The following example is in C and written for Quick C for Windows:

Stacking.C

```
#include <windows.h>
long far pascal typecheck (long a, float b, short far *c, char far *buff)
{
short retcode;
a = a * 3;
retcode = MessageBox(NULL, "I am in the DLL", "BOX", MB_OK);
```

```
return (a);
}
```

Stacking.DEF

```
LIBRARY      STACKING
EXETYPE     WINDOWS
STUB        'winstub.exe'
STACKSIZE   5120
HEAPSIZE    1024
DATA PRELOAD MOVEABLE SINGLE        ; ADD THESE TWO LINES
CODE PRELOAD MOVEABLE DISCARDABLE   ; TO AVOID WARNINGS.
EXPORTS
    typecheck   @1
     WEP           @2
```

Add the following code to the general Declarations module in a Visual Basic for Windows form:

```
Declare Function typecheck Lib "d\stacking.dll" (ByVal a As Long,
      ByVal b As Single, c As Integer, ByVal s As String) As Long
```

Note: The above declaration must be placed on one line.

In the Form_Click event:

```
Sub Form_Click ()
Dim a As Long   ' Explicitly type the variables.
Dim b As Single
Dim c As Integer
Dim s As String
a = 3                      ' Initialize the variables.
b = 4.5
c = 6
s = "Hello there! We've been waiting for you!"
Print typecheck(a, b, c, s)
End Sub
```

Running the program as written above will not generate the error. Now add the ByVal keyword before the variable named c in the Visual Basic for Windows Declaration. Run the program. Note that the MessageBox function pops a box first, and then the error box pops up indicating that Visual Basic for Windows checks the stack upon return to see if it has been correctly adjusted. Because the DLL expected a 4-byte pointer and received a 2-byte value, the stack has not adjusted back to the initial frame.

As another test, first remove the ByVal keyword before the variable 'c' that you added in the previous test. Declare the parameter 'a As Any' instead of As Long. Change the type of the variable 'a' in the Form_Click to Integer. Run the program again. Using As Any turns off type checking by Visual Basic for Windows. Because the program passed an integer ByVal instead of the long that the DLL expected, the stack frame is off and the error is generated.

Reference(s):

"Microsoft BASIC 7.0: Programmer's Guide" for versions 7.0 and 7.1, pages 423-426

Diagnosing "Error in loading DLL" with LoadLibrary

Article ID: Q90753

Summary:

The error "Error in loading DLL" (code 48) occurs when you call a dynamic-link library (DLL) procedure and the file specified in the procedure's Declare statement cannot be loaded. You can use the Microsoft Windows API function LoadLibrary to find out more specific information about why a DLL fails to load.

More Information:

The API function LoadLibrary loads a DLL and returns either a handle or an error code. If the return value is less than 32, it indicates one of the errors listed below. A return value greater than or equal to 32 indicates success and you should call the FreeLibrary function to unload the library.

LoadLibrary Error Codes
---

0 System was out of memory, executable file was corrupt, or relocations were invalid.

2 File was not found.

3 Path was not found.

5 Attempt was made to dynamically link to a task, or there was a sharing or network-protection error.

6 Library required separate data segments for each task.

8 There was insufficient memory to start the application.

10 Windows version was incorrect.

11 Executable file was invalid. Either it was not a Windows application or there was an error in the .EXE image.

12 Application was designed for a different operating system.

13 Application was designed for MS-DOS 4.0.

14 Type of executable file was unknown.

15 Attempt was made to load a real-mode application (developed for an earlier version of Windows).

16 Attempt was made to load a second instance of an executable file containing multiple data segments that were not marked read-only.

19 Attempt was made to load a compressed executable file. The file must be decompressed before it can be loaded.

20 Dynamic-link library (DLL) file was invalid. One of the DLLs required to run this application was corrupt.

21 Application requires Microsoft Windows 32-bit extensions.

Steps to Create Example Program

The following program demonstrates how to call LoadLibrary to load a library and display a resulting error code.

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Enter the following code into the general declarations section:

```
Declare Function LoadLibrary Lib "kernel" (ByVal f$) As Integer
Declare Sub FreeLibrary Lib "Kernel" (ByVal h As Integer)
```

3. Enter the following code into the Form Click event handler:

```
Sub Form_Click ()
   Dim hInst As Integer
   ' Enter the name of your DLL file inside the quotes below.
   ' The file WIN.COM is not a valid DLL and demonstrates an error.
   hInst = LoadLibrary("win.com")
   If hInst > 32 Then
      MsgBox "LoadLibrary success"
      FreeLibrary (hInst)
   Else
      MsgBox "LoadLibrary error " + Format$(hInst)
   End If
End Sub
```

4. Press the F5 key to run the program, then click on Form1. The program displays the error code returned from LoadLibrary. Look up this error code in the list of errors above to find an explanation.

How to Invoke Search in Windows Help from VB Program

Article ID: Q86771

Summary:

You can invoke the Search feature of the Windows Help engine from a Visual Basic program. To do this, call the Windows API function WinHelp and pass the constant HELP_PARTIALKEY (&H105) as the wCommand parameter and any string that is a NON-valid topic as the dwData parameter.

More Information:

Steps to Reproduce Behavior
---

1. Run Visual Basic, or from the File menu choose New Project (Press ALT, F, N) if Visual Basic is already running. Form1 will be created by default.

2. In GLOBAL.BAS, add the following code:

```
Global Const HELP_PARTIALKEY = &H105

Declare Function WinHelp Lib "User"(ByVal hWnd As Integer,
                                    ByVal lpHelpFile As String,
                                    ByVal wCommand As Integer,
                                    ByVal dwData As Any) As Integer
' Note: the declaration needs to be all on one line.
```

3. In the Form1 Click event procedure, add the following code:

```
Sub Form_Click ()
   DummyVal$ = " "
   Temp% = WinHelp(Form1.hWnd,
                   "c:\Windows\winhelp.hlp",
                   HELP_PARTIALKEY,
                  DummyVal$)
   ' Note: The function call must be all on one line.
End Sub
```

4. Press F5 to run this example, then click on the form.

How to Use Windows WNetAddConnection in Visual Basic

Article ID: Q94679

Summary:

Windows version 3.1 provides a new API Call, WNetAddConnection, which will redirect a local device to a shared resource or network server.

WNetAddConnection requires the name of the local device, the name of the network resource, and the password necessary to use that resource.

This article explains in detail the arguments and potential error messages for the Windows version 3.1 WNetAddConnection function call.

More Information:

To use WNetAddConnection within a Visual Basic application, declare the WNetAddConnection function in the General Declarations Section of your code window. (In Visual Basic version 1.0 you can also put the declaration in the Global Module.) Declare the function as follows:

```
Declare Function WnetAddConnection% Lib "user" (ByVal lpszNetPath As Any,
   ByVal lpszPassword As Any,
   ByVal lpszLocalName As Any)
```

Here are definitions for the formal parameters:

Formal Parameter	Definition
lpszNetPath	Points to a null-terminated string specifying the shared device or remote server.
lpszPassword	Points to a null-terminated string specifying the network password for the given device or server.
lpszLocalName	Points to a null-terminated string specifying the local drive or device to be redirected. All lpszLocalName strings (such as LPT1) are case independent. Only the drive names A through Z and device names LPT1 through LPT3 are used.

Below are the possible return values as defined on page 990 of the Microsoft Windows version 3.1 Programmer's Reference:

Value	(Hex Value)	Meaning
WN_SUCCESS	(&H0)	Function was successful.
WN_NOT_SUPPORTED	(&H1)	Function was not supported.
WN_OUT_OF_MEMORY	(&HB)	System was out of memory.
WN_NET_ERROR	(&H2)	An error occurred on the network.
WN_BAD_POINTER	(&H4)	Pointer was invalid.
WN_BAD_NETNAME	(&H32)	Network resource name was invalid.
WN_BAD_LOCALNAME	(&H33)	Local device name was invalid.
WN_BAD_PASSWORD	(&H6)	Password was invalid.
WN_ACCESS_DENIED	(&H7)	A security violation occurred.
WN_ALREADY_CONNECTED	(&H34)	Local device was already connected to a remote resource.

Below is an example of how to redirect a local device to a network resource:

1. Start Visual Basic (VB.EXE). Form1 is created by default.

2. Create the following controls with the indicated properties on Form1:

Default Name	Caption	CtlName
Text1	(Not applicable)	NetPath
Text2	(Not applicable)	Password
Command1	&Connect	Connect
Drive1	(Not applicable)	Drive1

3. Add the following code to the general declaration section of Form1. Note that the Declare Function statement appears as four lines for readability, but you should enter it as a single line:

```
Declare Function WnetAddConnection% Lib "user"
   (ByVal lpszNetPath as Any,
   ByVal lpszPassword as Any,
   ByVal lpszLocalName as Any) ' Put entire Declare on a single line.

Const WN_Success = &H0
Const WN_Not_Supported = &H1
Const WN_Net_Error = &H2
Const WN_Bad_Pointer = &H4
Const WN_Bad_NetName = &H32
Const WN_Bad_Password = &H6
Const WN_Bad_Localname = &H33
Const WN_Access_Denied = &H7
Const WN_Out_Of_Memory = &HB
Const WN_Already_Connected = &H34
```

If you're using Visual Basic version 1.0, add the following to the general declarations also:

```
Const True = -1
Const False = 0
```

4. Add the following code to the procedure Connect_Click:

```
Sub Connect_Click ()

  ServerText$ = UCase$(NetPath.Text) + Chr$(0) ' Network resource name
  PasswordText$ = Password.Text + Chr$(0) ' Password for the resource
  driveletter$ = "N:" + Chr$(0) ' Substitute your own drive letter

  Succeed% = WnetAddConnection(ServerText$, PasswordText$, driveletter$)

  If IsSuccess(Succeed%, msg$) = True Then ' Call Function to parse
                                           ' potential error messages.
     Drive1.Refresh
     NetPath.Text = "" ' Reset the contents following connection
  Else
     MsgBox msg$
```

```
  End If

End Sub
```

5. Create a Sub within the (Declarations) section of the Code window and add the following code:

```
Function IsSuccess% (ReturnCode%, ErrMsg$)

If ReturnCode% = WN_Success Then
   IsSuccess% = True
Else
   IsSuccess% = False
   Select Case ReturnCode%

      Case WN_Success:
         Drive1.Refresh
      Case WN_Not_Supported:
         msg$ = "Function is not supported."
      Case Wn_Out_Of_Memory:
         msg$ = "Out of Memory."
      Case WN_Net_Error:
         msg$ = "An error occurred on the network."
      Case WN_Bad_Pointer:
         msg$ = "The Pointer was Invalid."
      Case WN_Bad_NetName:
         msg$ = "Invalid Network Resource Name."
      Case WN_Bad_Password:
         msg$ = "The Password was Invalid."
      Case WN_Bad_Localname:
         msg$ = "The local device name was invalid."
      Case WN_Access_Denied:
         msg$ = "A security violation occurred."
      Case WN_Already_Connected:
         msg$ = "The local device was connected to a remote resource."
      Case Else:
         msg$ = "Unrecognized Error " + Str$(ReturnCode%) + "."

   End Select
End If

End Function
```

6. Run the program. Type in the name of a network resource in the edit box and press the Connect button. The drive box will be updated with the new resource if the call was successful.

How to Invoke GetSystemMetrics Windows API Function from VB

Article ID: Q77061

Summary:

The Windows API GetSystemMetrics function can return useful information about the Windows system. GetSystemMetrics can be called directly from Visual Basic for Windows or from the custom Control Development Kit (CDK) to get system metrics for a particular display adapter, retrieve information about the Windows debug mode, or retrieve information about a mouse configuration.

The Visual Basic for Windows CDK is shipped as part of the Professional Edition of Microsoft Visual Basic versions 2.0 or 3.0. for Windows.

More Information:

The Windows GetSystemMetrics function call retrieves information about the system metrics. The system metrics are the widths and heights of various display elements of the particular window display. The GetSystemMetrics function can also return flags that indicate whether the current Windows version is a debugging version, whether a mouse is present, or whether the meaning of the left and right mouse button has been changed. System metrics depend on the system display, and may vary from display to display.

The Visual Basic for Windows declaration for GetSystemMetrics is:

```
Declare  Function GetSystemMetrics% Lib  "user"  (ByVal nIndex%)
```

The value nIndex% specifies the system measurement to be retrieved. All measurements are in pixels.

The value returned from the GetSystemMetrics% function specifies the system metrics.

Below is a sample call to determine if the present version of Windows is a debugging version:

```
ScaleMode = 3  ' Select pixel.
Print "Debugging version :  ; GetSystemMetrics(SM_DEBUG).
```

The constants and meaning for nIndex% are as follows:

```
Constant Name(Value)  Description
--------------------  -----------

SM_CXSCREEN(0).........Width of screen
SM_CYSCREEN(1).........Height of screen
SM_CXFRAME(32).........Width of window frame that can be sized
SM_CYFRAME(33).........Height of window frame that can be sized
SM_CXVSCROLL(2).......Width of arrow bitmap on vertical scroll bar
SM_CYVSCROL(20).......Height of arrow bitmap on vertical scroll bar
SM_CXHSCROLL(21)......Width of arrow bitmap on horizontal scroll bar
SM_CYHSCROLL(3).......Height of arrow bitmap on horizontal scroll
                      bar
SM_CYCAPTION(4).......Height of caption
```

```
SM_CXBORDER(5).........Width of window frame that cannot be sized
SM_CYBORDER(6).........Height of window frame that cannot be sized
SM_CXDLGFRAME(7).......Width of frame when window has WS_DLGFRAME
                       style
SM_CYDLGFRAME(8).......Height of frame when window has WS_DLGFRAME
                       style
SM_CXHTHUMB(10)........Width of thumb on horizontal scroll bar
SM_CYHTHUMB(9).........Height of thumb on horizontal scroll bar
SM_CXICON(11)..........Width of icon
SM_CYICON(12)..........Height of icon
SM_CXCURSOR(13)........Width of cursor
SM_CYCURSOR(14)........Height of cursor
SM_CYMENU(15)..........Height of single-line menu
SM_CXFULLSCREEN(16)...Width of window client area for full-screen
                       window
SM_CYFULLSCREEN(17)...Height of window client area for full-screen
                       window (height - caption)
SM_CYKANJIWINDOW(18)..Height of Kanji window
SM_CXMINTRACK(34).....Minimum tracking width of window
SM_CYMINTRACK(35).....Minimum tracking height of window
SM_CXMIN(28)...........Minimum width of window
SM_CYMIN(29)...........Minimum width of window
SM_CXSIZE(30)..........Width of bitmaps contained in the title bar
SM_CYSIZE(31)..........Height of bitmaps contained in the title bar
SM_MOUSEPRESENT(19)...Mouse present
SM_DEBUG(22)...........Nonzero if Windows debug version
```

How VB Can Get Windows Status Information via API Calls

Article ID: Q84556

Summary:

The Visual Basic for Windows program example below demonstrates how you can obtain system status information similar to the information displayed in the Windows Program Manager About box. The example program displays the following information using the Windows API function(s) indicated:

- The Windows version number with GetVersion
- The kind of CPU (80286, 80386, or 80486) and whether a math coprocessor is present with GetWinFlags
- Whether Windows is running in enhanced mode or standard mode with GetWinFlags
- The amount of free memory with GetFreeSpace and GlobalCompact
- The percentage of free system resources with SystemHeapInfo

Note: The API function SystemHeapInfo is new to Windows version 3.1 and is not available in Windows, version 3.0. All other API functions listed above are available in both Windows versions 3.0 or 3.1.

More Information:

Steps to Create Example Program
---

1. Run Visual Basic for Windows, or if Visual Basic for Windows is already running, choose New Project from the File menu (press ALT, F, N). Form1 will be created by default.

2. From the File menu, choose Add Module (press ALT, F, M). Module 1 is created by default (In Visual Basic version 1.0 for Windows, this step is unnecessary).

3. Enter the following code into the general declarations section of a code module (In Visual Basic version 1.0 for Windows, place the following in the Global module):

```
' Constants for GetWinFlags.
Global Const WF_CPU286 = &H2
Global Const WF_CPU386 = &H4
Global Const WF_CPU486 = &H8
Global Const WF_80x87 = &H400
Global Const WF_STANDARD = &H10
Global Const WF_ENHANCED = &H20

' Type for SystemHeapInfo.
Type SYSHEAPINFO
    dwSize As Long
    wUserFreePercent As Integer
    wGDIFreePercent As Integer
    hUserSegment As Integer
```

```
    hGDISegment As Integer
End Type

Declare Function GetVersion Lib "Kernel" () As Integer
Declare Function GetWinFlags Lib "Kernel" () As Long
Declare Function GetFreeSpace Lib "Kernel" (ByVal wFlags As Integer)
   As Long
Declare Function GlobalCompact Lib "Kernel" (ByVal dwMinFree As Long)
   As Long
Declare Function SystemHeapInfo Lib "toolhelp.dll" (shi As
   SYSHEAPINFO) As Integer
' Each Declare statement above must appear on a single line.
```

4. Enter the following code into the Form_Load procedure of Form1:

```
Sub Form_Load ()
    Dim msg As String         ' Status information.
    Dim nl As String          ' New-line.
    nl = Chr$(13) + Chr$(10)  ' New-line.

    Show
    mp% = MousePointer
    MousePointer = 11   ' Hourglass.

    ' Get operating system version.
    ver% = GetVersion()
    ver_major$ = Format$(ver% And &HFF)
    ver_minor$ = Format$(ver% \ &H100, "00")
    msg = msg + "Microsoft Windows version "
    msg = msg + ver_major$ + "." + ver_minor$ + nl

    ' Get CPU kind and operating mode.
    msg = msg + "CPU: "
    status& = GetWinFlags()
    If status& And WF_CPU286 Then msg = msg + "80286"
    If status& And WF_CPU386 Then msg = msg + "80386"
    If status& And WF_CPU486 Then msg = msg + "80486"
    If status& And WF_80x87 Then msg = msg + " with 80x87"
    msg = msg + nl
    msg = msg + "Mode: "
    If status& And WF_STANDARD Then msg = msg + "Standard" + nl
    If status& And WF_ENHANCED Then msg = msg + "Enhanced" + nl

    ' Get free memory.
    memory& = GetFreeSpace(0)
    msg = msg + "Memory free: "
    msg = msg + Format$(memory& \ 1024, "###,###,###") + "K" + nl
    memory& = GlobalCompact(&HFFFFFFFF)
    msg = msg + "Largest free block: "
    msg = msg + Format$(memory& \ 1024, "###,###,###") + "K" + nl

    ' Get free system resources.
    ' The API SystemHeapInfo became available in Windows version 3.1.
    msg = msg + "System resources: "
    If ver% >= &H310 Then
        Dim shi As SYSHEAPINFO
        shi.dwSize = Len(shi)
```

```
        If SystemHeapInfo(shi) Then
            If shi.wUserFreePercent < shi.wGDIFreePercent Then
                msg = msg + Format$(shi.wUserFreePercent) + "%"
            Else
                msg = msg + Format$(shi.wGDIFreePercent) + "%"
            End If
        End If
    Else
        msg = msg + "n/a"
    End If

    MsgBox msg, 0, "About " + Caption
    MousePointer = mp%
End Sub
```

5. Press the F5 key to run the program.

How to Add a Horizontal Scroll Bar to Visual Basic List Box

Article ID: Q80190

Summary:

The normal list box that comes with Visual Basic for Windows does not have a horizontal scroll bar. This can be a problem when the item in a list box extends past the boundaries of the list box. To add a horizontal scroll bar to the control, you can call the Windows API SendMessage function with the LB_SETHORIZONTALEXTENT (WM_USER + 21) constant.

More Information:

To add a horizontal scroll bar to a list box, perform a SendMessage function call with the LB_SETHORIZONTALEXTENT constant.

This message sets the width in pixels by which a list box can scroll horizontally. If the size of the list box is smaller than this value, the horizontal scroll bar will horizontally scroll items in the list box. If the list box is large as or larger than this value, the horizontal scroll bar is disabled.

The parameters for the SendMessage function are as follows:

```
SendMessage(hWnd%, LB_SETHORIZONTALEXTENT, wParam%, lParam&)
------------------------------------------------------------

   hWnd%   - Handle to the list box
   wParam% - Specifies the number of pixels by which the list
             box can be scrolled
   lParam% - Is not used
```

To make a program example that will only allow the user to scroll a specified distance, create a form with the following controls:

Control	Name (use CtlName in Visual Basic 1.0 for Windows)
Command button	Command1
List box	List1

Add the following code in the described locations in your code:

```
'======== General Declarations for Form1  ==================
Declare Function SendMessage& Lib "user" (
        ByVal hWnd%,
        ByVal wMsg%,
        ByVal wParam%,
        ByVal lParam&)
Declare Function GetFocus Lib "User" () as Integer

'======== Form1 ======================
'Note: All commands must appear on only one line.

Sub Command1_Click ()
 Const LB_SETHORIZONTALEXTENT = &H400 + 21
 Const NUL = &O0
   ' wParam is in PIXEL(3).
```

```
    ScaleMode = 3

    ' Get the handle.
    List1.SetFocus
    ListHwnd% = GetFocus()

    ' This string will show up initially.
    ListString1$ = "Derek is a great "

    ' You can scroll to see this portion.
    ListString2$ = "little boy "

    ' You cannot scroll to see this string.
    ListString3$ = "but can be a problem sometimes"

    ExtraPixels% = TextWidth(ListString2$)
    BoxWidth% = TextWidth(ListString1$)

    ' Resize the text box.
    List1.Move List1.Left, List1.Top, BoxWidth%

    ' Add the scroll bar.
    X& = SendMessage(ListHwnd%, LB_SETHORIZONTALEXTENT,
                     BoxWidth% + ExtraPixels%, NUL)

    ' Add the example string to the list box.
    List1.AddItem ListString1$ + ListString2$ + ListString3$
End Sub
```

How to Create a Flashing Title Bar on a Visual Basic Form

Article ID: Q71280

Summary:

When calling a Windows API function call, you can create a flashing window title bar on the present form or any other form for which you know the handle.

More Information:

Visual Basic for Windows has the ability to flash the title bar on any other form if you can get the handle to that form. The function FlashWindow flashes the specified window once. Flashing a window means changing the appearance of its caption bar, as if the window were changing from inactive to active status, or vice versa. (An inactive caption bar changes to an active caption bar; an active caption bar changes to an inactive caption bar.)

Typically, a window is flashed to inform the user that the window requires attention when that window does not currently have the input focus.

The function FlashWindow is defined as

```
FlashWindow(hWnd%, bInvert%)
```

where:

```
hWnd%     - Identifies the window to be flashed. The window can be
            either open or iconic.
bInvert%  - Specifies whether the window is to be flashed or
            returned to its original state. The window is flashed
            from one state to the other if the bInvert parameter is
            nonzero. If the bInvert parameter is zero, the window
            is returned to its original state (either active or
            inactive).
```

FlashWindow returns a value that specifies the window's state before the call to the FlashWindow function. It is nonzero if the window was active before the call; otherwise, it is zero.

The following section describes how to flash a form while that form does not have the focus:

1. Create two forms called Form1 and Form2.

2. On Form1, create a timer control and set the Interval Property to 1000. Also set the Enabled Property to FALSE.

3. Within the general-declarations section of Form1, declare the FlashWindow function as follows:

```
' The following Declare statement must appear on one line.
Declare Function FlashWindow% Lib "user" (ByVal hWnd%,
                                          ByVal bInvert%)
```

4. In Visual Basic version 1.0 for Windows, define the following

constants in the declarations section:

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

5. In the Form_Load event procedure, add the following code:

```
Sub Form_Load ()
   Form2.Show
End Sub
```

6. In the Sub Timer1_Timer () procedure of Form1, add the following code:

```
Sub Timer1_Timer ()
   Succ% = FlashWindow(Form1.hWnd, 1)
End Sub
```

7. In the GotFocus event procedure of Form1, create the following code:

```
Sub Form_GotFocus ()
  Timer1.Enabled =  False
End Sub
```

8. In the Click event for Form2, add the following code:

```
Sub Form_Click ()
   Form1.Timer1.Enabled = True
End Sub
```

9. Run the program. Form1 will be in the foreground with Form2 in the background. Click anywhere on Form2; Form1's Caption Bar will flash until you click on Form1.

How to Set Windows System Colors Using API and Visual Basic

Article ID: Q82158

Summary:

This article describes how to use the GetSysColor and SetSysColors API functions to set the system colors for various parts of the display in Microsoft Windows. This allows you to change the Windows display programmatically, instead of using the Windows Control Panel.

More Information:

Windows maintains an internal array of 19 color values that it uses to paint the different parts of the Windows display. Changing any of these values will affect all windows for all applications running under Windows. Note that the SetSysColors routine only changes the internal system list. This means that any changes made using SetSysColors will only be valid for the current Windows session. To make these changes permanent, you need to change the [COLORS] section of the Windows initialization file, WIN.INI.

To use the GetSysColor and SetSysColors functions within a Visual Basic for Window application, you must first declare them in the Declarations section of your Code window.

Declare the Function statement as follows:

```
Declare Function GetSysColor Lib "User" (ByVal nIndex%) As Long

Declare Sub SetSysColors Lib "User" (ByVal nChanges%,
                                     lpSysColor%,
                                     lpColorValues&)
```

Note: Each Declare statement above must be written on one line.

The parameters are defined as follows:

Parameter	Definition
nIndex%	Specifies the display element whose color is to be retrieved. See the list below to find the index value for the corresponding display element.
nChanges%	Specifies the number of system colors to be changed.
lpSysColor%	Identifies the array of integer indexes that specify the elements to be changed.
lpColorValues&	Identifies the array of long integers that contain the new RGB color values for each element to be changed.

The following system color indexes are defined using the predefined constants found in the WINDOWS.H file supplied with the Microsoft Windows Software Development Kit (SDK). The corresponding value is

the value placed in the lpSysColor% array.

List of System Color Indexes

Windows.H Definition	Value	Description
COLOR_SCROLLBAR	0	Scroll-bar gray area
COLOR_BACKGROUND	1	Desktop
COLOR_ACTIVECAPTION	2	Active window caption
COLOR_INACTIVECAPTION	3	Inactive window caption
COLOR_MENU	4	Menu background
COLOR_WINDOW	5	Window background
COLOR_WINDOWFRAME	6	Window frame
COLOR_MENUTEXT	7	Text in menus
COLOR_WINDOWTEXT	8	Text in windows
COLOR_CAPTIONTEXT	9	Text in caption, size box, scroll bar arrow box
COLOR_ACTIVEBORDER	10	Active window border
COLOR_INACTIVEBORDER	11	Inactive window border
COLOR_APPWORKSPACE	12	Background color of multiple document interface (MDI) applications
COLOR_HIGHLIGHT	13	Items selected item in a control
COLOR_HIGHLIGHTTEXT	14	Text of item selected in a control
COLOR_BTNFACE	15	Face shading on push button
COLOR_BTNSHADOW	16	Edge shading on push button
COLOR_GRAYTEXT	17	Grayed (disabled) text. This color is set to 0 if the current display driver does not support a solid gray color.
COLOR_BTNTEXT	18	Text on push buttons

The following is an example of how to set the system colors for different parts of the Windows display:

1. Start Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Create the following controls for Form1:

Control	Name	Property Setting
Command button	Command1	Caption = "Change all Colors"
Command button	Command2	Caption = "Change selected Colors"

   (In Visual Basic version 1.0 for Windows, set the CtlName Property for the above objects instead of the Name property.)

3. Add the following code to the general Declarations section of Form1:

```
Declare Function GetSysColor Lib "User" (ByVal nIndex%) As Long

Declare Sub SetSysColors Lib "User" (ByVal nChanges%,
                                     lpSysColor%,
                                     lpColorValues&)
' Note: The above declaration must be on one line.

 Const COLOR_BACKGROUND = 1
 Const COLOR_ACTIVECAPTION = 2
 Const COLOR_WINDOWFRAME = 6

 Dim SavedColors(18) As Long
```

4. Add the following code to the Form_Load event procedure of Form1:

```
Sub Form_Load ()

' ** Save current system colors.
   For i% = 0 To 18
      SavedColors(i%) = GetSysColor(i%)
   Next i%

End Sub
```

5. Add the following code to the Form_Unload event procedure of Form1:

```
Sub Form1_Unload ()

' ** Restore system colors.
   ReDim IndexArray(18) As Integer
   For i% = 0 To 18
      IndexArray(i%) = i%
   Next i%
   SetSysColors 19, IndexArray(0), SavedColors(0)

End Sub
```

6. Add the following code to the Command1_Click event procedure of Form1:

```
Sub Command1_Click ()

' ** Change all display elements.
   ReDim NewColors(18) As Long
   ReDim IndexArray(18) As Integer
   For i% = 0 to 18
      NewColors(i%) = QBColor(Int(16 * Rnd))
      IndexArray(i%) = i%
   Next i%
   SetSysColors 19, IndexArray(0), NewColors(0)

End Sub
```

7. Add the following code to the Command2_Click event procedure of Form1:

```
Sub Command2_Click ()
```

```
    ' ** Change desktop, window frames, and active caption.
        ReDim NewColors(18) As Long
       ReDim IndexArray(18) As Integer
       For i% = 0 to 18
          NewColors(i%) = QBColor(Int(16 * Rnd))
          IndexArray(i%) = i%
       Next i%
       SetSysColors 19, IndexArray(0), NewColors(0)

    End Sub
```

8. From the Run menu, choose Start, or press the F5 key, to run the program.

Choosing the Change All Colors button will cause all the different parts of the Windows display to be assigned a randomly generated color. Choosing the Change Selected Elements button will cause only the desktop, active window caption, and window frames to be assigned a random color. To restore the original system colors, double-click the Control-menu box to end the application.

Creating TOPMOST or "Floating" Window in Visual Basic

Article ID: Q84251

Summary:

You can create a "floating" window such as that used for the Microsoft Windows version 3.1 Clock by using the SetWindowPos Windows API call.

More Information:

A floating (or TOPMOST) window is a window that remains constantly above all other windows, even when it is not active. Examples of floating windows are the Find dialog box in WRITE.EXE, and CLOCK.EXE (when Always on Top is selected from the Control menu).

There are two methods to produce windows that "hover" or "float," one of which is possible in Visual Basic for Windows. This method is described below:

Call SetWindowPos, specifying an existing non-topmost window and HWND_TOPMOST as the value for the second parameter (hwndInsertAfter):

Use the following declarations:

```
Declare Function SetWindowPos Lib "user" (ByVal h%, ByVal hb%,
        ByVal x%, ByVal y%, ByVal cx%, ByVal cy%, ByVal f%) As Integer
' The above Declare statement must appear on one line.

Global Const SWP_NOMOVE = 2
Global Const SWP_NOSIZE = 1
Global Const FLAGS = SWP_NOMOVE Or SWP_NOSIZE
Global Const HWND_TOPMOST = -1
Global Const HWND_NOTOPMOST = -2
```

To set the form XXXX to TOPMOST, use the following code:

```
success% = SetWindowPos (XXXX.hWnd, HWND_TOPMOST, 0, 0, 0, 0, FLAGS)
REM success% <> 0 When Successful
```

To reset the form XXXX to NON-TOPMOST, use the following code:

```
success% = SetWindowPos (XXXX.hWnd, HWND_NOTOPMOST, 0, 0, 0, 0, FLAGS)
REM success% <> 0 When Successful
```

Note: This attribute was introduced in Windows, version 3.1, so remember to make a GetVersion() API call to determine whether the application is running under Windows, version 3.1.

Example of How to Read and Write Visual Basic Arrays to Disk

Article ID: Q77317

Summary:

Microsoft Visual Basic for Windows does not provide a command to read or write an entire array all at once to a disk file. Using Visual Basic for Windows alone, you must transfer each element of the array to the disk. However, using two Windows API functions, _lread and _lwrite, you can save an entire array to disk in one statement with arrays less then 64K, or you can use _hread and _hwrite for arrays greater than 64k. The example below demonstrates how to use the _lread and _lwrite functions with Visual Basic for Windows.

More Information:

The ReadArray and WriteArray functions provided below allow you to read and write a Visual Basic for Windows array to or from a disk file. These functions will work with arrays of Integers, Longs, Singles, Doubles, Currency, and user-defined types, but not with variable-length strings (as an array or as a member of a user-defined type) or variants. These functions can work with fixed length strings when the strings are a member of a user-defined type. Although, _lread and _lwrite do not handle the huge arrays supported by Visual Basic versions 2.0 and later for Windows, the _hread and _hwrite Windows API functions can read and write huge arrays to and from disk.

The two functions, ReadArray and WriteArray, require two parameters: the array to be transferred, and the Visual Basic for Windows file number to be written to or read from. The functions also return the number of bytes transferred, or -1 when an error occurs with the API function. The file number is the Visual Basic for Windows file number of a file that has already been opened with the Open statement, and will be used in the Visual Basic for Windows Close statement.

The following function examples use a user-defined type named "Mytype". An example of this type is as follows:

```
   Type MyType
        Field1 As String * 10
        Field2 As Integer
        Field3 As Long
        Field4 As Single
        Field5 As Double
        Field6 As Currency
   End Type
```

Declarations of API Functions

```
DefInt A-Z
' Each Declare statement must appear on one line:
Declare Function fWrite Lib "kernel" Alias "_lwrite" (ByVal hFile,
   lpBuff As Any, ByVal wBytes)
Declare Function fRead Lib "kernel" Alias "_lread" (ByVal hFile,
   lpBuff As Any, ByVal wBytes)
```

```
Function: ReadArray
-------------------

Function ReadArray (An_Array() As MyType, VBFileNumber As Integer) As Long

   Dim ArraySize As Long
   Dim DOSFileHandle As Integer
   Dim ReadFromDisk As Integer

   ArraySize = Abs(UBound(An_Array) - LBound(An_Array)) + 1
   ArraySize = ArraySize * Len(An_Array(LBound(An_Array)))

   If ArraySize > 32767 Then
      ReadFromDisk = ArraySize - 2 ^ 15
      ReadFromDisk = WriteToDisk * -1
   Else
      ReadFromDisk = ArraySize
   End If

   DOSFileHandle = FileAttr(VBFileNumber, 2)
   ApiErr=fRead(DOSFileHandle,An_Array(LBound(An_Array)),ReadFromDisk)

   ReadArray = ApiErr
End Function

Function: WriteArray
--------------------

Function WriteArray (An_Array() As MyType, VBFileNumber As Integer) As Long

   Dim ArraySize As Long
   Dim DOSFileHandle As Integer
   Dim WriteToDisk As Integer

   ArraySize = UBound(An_Array) - LBound(An_Array) + 1
   ArraySize = ArraySize * Len(An_Array(LBound(An_Array)))

   If ArraySize > 32767 Then
      WriteToDisk = ArraySize - 2 ^ 15
      WriteToDisk = WriteToDisk * -1
   Else
      WriteToDisk = ArraySize
   End If

   DOSFileHandle = FileAttr(VBFileNumber, 2)
   ApiErr=fWrite(DOSFileHandle,An_Array(LBound(An_Array)),WriteToDisk)

   WriteArray = ApiErr
End Function
```

The following are the function header changes to allow the ReadArray and WriteArray functions to work with different data types (Integer, Long, Single, Double, Currency, and user-defined type). Each Function statement must be on a single line:

```
Function ReadArray  (An_Array() As Integer, VBFileNumber As Integer)
   As Long
```

```
Function WriteArray (An_Array() As Integer, VBFileNumber As Integer)
   As Long

Function ReadArray  (An_Array() As Long, VBFileNumber As Integer) As
   Long
Function WriteArray (An_Array() As Long, VBFileNumber As Integer) As
   Long

Function ReadArray  (An_Array() As Single, VBFileNumber As Integer) As
   Long
Function WriteArray (An_Array() As Single, VBFileNumber As Integer) As
   Long

Function ReadArray  (An_Array() As Double, VBFileNumber As Integer) As
   Long
Function WriteArray (An_Array() As Double, VBFileNumber As Integer) As
   Long

Function ReadArray  (An_Array() As Currency, VBFileNumber As Integer)
   As Long
Function WriteArray (An_Array() As Currency, VBFileNumber As Integer)
   As Long
```

How to Determine Display State of a VB Form, Modal or Modeless

Article ID: Q77316

Summary:

The Show method in the Visual Basic for Windows language can display a form either as modal or modeless. No direct support exists in the language to determine the display state of the form without maintaining global variables that contain the display state of the form. However, the Windows API function GetWindowLong can be used to check the display state of the form.

More Information:

When Visual Basic for Windows displays a modal form (.Show 1), all other forms will be modified to contain the Window Style WS_DISABLED. The Windows API function GetWindowLong can be used to return the Window Style of another form to check for the WS_DISABLED style.

The following code demonstrates this process:

Add the following to the General Declarations section of Form1 and Form2:

```
DefInt A-Z
Global Const GWL_STYLE = (-16)
Global Const WS_DISABLED = &H8000000
Declare Function GetWindowLong& Lib "user" (ByVal hWnd, ByVal nIndex)
```

Form1.Frm

```
Sub Form_Click ()
  ' Flip between "Modeless" and "Modal" display states.
  Static ShowStyle
  Unload form2
  form2.Show ShowStyle
  ShowStyle = (ShowStyle + 1) Mod 2
End Sub
```

Form2.Frm

```
Sub Form_Paint ()
  ' Get the Window Style for Form1.
   WinStyle& = GetWindowLong(Form1.hWnd, GWL_STYLE)
   If WinStyle& And WS_DISABLED Then
      ' The WS_DISABLED style is set on "FORM1" when "FORM2"
      ' is displayed with the Modal flag (Show 1).
      Print "Modal    - Show 1"
   Else
      ' The WS_DISABLED style is not set on "FORM1" when "FORM2"
      ' is displayed with the Modeless flag (Show or Show 0).
      Print "Modeless - Show"
   End If
End Sub
```

How to Determine the Number of VB Applications Running at Once

Article ID: Q84836

Summary:

To determine the total number of Microsoft Visual Basic for Windows applications running at any given time, you can use the Microsoft Windows API functions GetModuleHandle and GetModuleUsage.

More Information:

The following code fragment demonstrates a technique to find the total number of Visual Basic for Windows applications currently executing by determining the number of instances of the Visual Basic run-time module (VBRUN100.DLL) with the Windows API functions GetModuleHandle and GetModuleUsage. Remember that Visual Basic for Windows itself is not counted; only applications created with Visual Basic for Windows are included.

Steps to Create Example Program
---

1. Start several Visual Basic for Windows applications and leave them running.

2. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

3. Enter the following Windows API function declarations into the General Declarations section of Form1:

   ```
   Declare Function GetModuleUsage% Lib "kernel" (ByVal hModule%)
   Declare Function GetModuleHandle% Lib "kernel" (ByVal FileName$)
   ```

4. Place a command button (Command1) on Form1. Double-click that button to open the Code window. In the Command1_Click procedure, add the following code:

   ```
   Sub Command1_Click ()
     msg$ = "Number of executing VB Apps: "

     hModule% = GetModuleHandle("VBRUN300.DLL")
       ' For Visual Basic versions 1.0 and 2.0 for Windows, use
       ' VBRun100.DLL and VBRun2.00.DLL respectively.
     nInstances% = GetModuleUsage(hModule%)

     msg$ = msg$ + Str$(nInstances%)
     MsgBox msg$
   End Sub
   ```

5. From the File menu, choose Make EXE File.

6. Press the F5 key to run the file.

7. Click on the command button.

A message box displays the total number of executing Visual Basic

for Windows applications.

Note: This program itself will count as one application.

How to Kill an Application with System Menu Using Visual Basic

Article ID: Q80186

Summary:

Visual Basic for Windows can use the Windows API SendMessage function to close any active window that has a system menu (referred to as control box within Visual Basic for Windows) with the Close option.

More Information:

You can use the Windows API SendMessage function to post a message to any window in the environment as long as the handle to the window is known. You can use the API FindWindow function to determine the handle associated with the window the user wants to close.

To create a program to close an occurrence of the Windows version 3.0 Calculator program, do the following:

1. Create a form called Form1.

2. Create two command buttons called Command1 and Command2.

3. Within the Command1 Click event, add the following code:

```
Sub Command1_Click()
   X% = Shell("Calc.exe")
End Sub
```

4. Within the Command2 Click event, add the following code:

```
Sub Command2_Click()
  Const NILL = 0&
  Const WM_SYSCOMMAND = &H112
  Const SC_CLOSE = &HF060

   lpClassName$ = "SciCalc"
   lpCaption$ = "Calculator"

   '* Determine the handle to the Calculator window.
   Handle = FindWindow(lpClassName$, lpCaption$)

   '* Post a message to Calc to end it's existence.
   X& = SendMessage(Handle, WM_SYSCOMMAND, SC_CLOSE, NILL)

End Sub
```

5. In the Declarations section, declare the following two API functions:

```
'* NOTE: Each Declare statement must appear on one line.
Declare Function FindWindow% Lib "user" (
        ByVal lpClassName As Any,
        ByVal lpCaption As Any)
Declare Function SendMessage& Lib "user" (
        ByVal hwnd%,
        ByVal wMsg%,
        ByVal wParam%,
        ByVal lParam As Long)
```

6. Run the program. A click on Command1 will bring up an instance of the Calculator program. A click on Command2 will close the window.

How to Set Focus to First VB .EXE Instance When Second Invoked

Article ID: Q84585

Summary:

This article describes how to set the focus to the first instance of a Visual Basic for Windows .EXE application when you attempt to invoke a second instance of the same application. This feature prevents multiple copies (instances) of the same program from running in memory.

More Information:

An example of this behavior is shown by the File Manager shipped with Windows. If the File Manager is already running and you try to start a second instance of it, the focus is simply shifted to the copy that is already running so that another occurrence is not started. By using the following function, you can achieve the same effect in a Visual Basic for Windows application.

Steps to Reproduce Behavior
---

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Place a command button (Command1) on Form1. Set the Caption property to END. In the Command1_Click event, put the keyword END as the only line of code.

3. Put the following declarations in either the global module or the general Declarations section of Form1. There are three declarations, and each must be on one line.

```
Declare Function FindWindow% Lib "user" (ByVal lpClassName As Any,
                 ByVal lpCaption As Any)
Declare Function ShowWindow% Lib "User" (ByVal Handle As Integer,
                 ByVal Cmd As Integer)
Declare Function SFocus% Lib "User" Alias "SetFocus" (ByVal Handle As
                 Integer)
```

4. Put the following code in the Form1.Load event:

```
Title$ = "Test Program"
 X% = CheckUnique(Title$)
 If X% = 0 Then
     End
 End If
 Form1.Caption= Title$
```

5. Create the following general function:

```
 Function CheckUnique (FormName As String) As Integer
     Dim Handle As Integer
     Handle = FindWindow(0&, FormName)

     If Handle = 0 Then
         ' -1 is a true value.
         CheckUnique = -1
```

```
        Else
            X% = ShowWindow(Handle, 1)
            X% = SFocus(Handle)
            ' 0 is a false value.
            CheckUnique = 0
        End If
    End Function
```

6. From the File menu, choose Make EXE File.

7. Press the F5 key to run the program.

If you try to launch a second occurrence of the program, it will simply give the focus to the first. If you try to launch a second occurrence while the first occurrence is minimized, it will restore the first occurrence and give it the focus. A second occurrence will not be loaded into Windows.

How to Create a System-Modal Program/Window in Visual Basic

Article ID: Q72674

Summary:

From a Microsoft Visual Basic for Windows program, you can disable the ability to switch to other Windows programs by calling the Windows API function SetSysModalWindow.

More Information:

Microsoft Windows is designed so that the user can switch between applications without terminating one program to run another program. There may be times when the program needs to take control of the entire environment and run from only one window, restricting the user from switching to any other application. An example of this is a simple security system, or a time-critical application that may need to go uninterrupted for long periods of time.

Passing the handle to the window through the argument of SetSysModalWindow will limit the user to that particular window. This will not allow the user to move to any other applications with the mouse or use ALT+ESC or CTRL+ESC to bring up the Task Manager. You can even remove the system menu if you do not want the user to exit through the ALT+F4 (Close) combination.

All child windows that are created by the system-modal window become system-modal windows. When the original window becomes active again, it is system-modal. To end the system-modal state, destroy the original system-modal window.

Care must be taken when using the SetSysModalWindow API from within the Visual Basic for Windows programming environment. Pressing CTRL+BREAK to get to the [break] mode leaves your modal form with no way to exit unless you restart your system. When using the SetSysModalWindow within the environment, be sure to exit your application by destroying the window with either the ALT+F4 in the system menu, or by some other means from within your running program.

To use the SetSysModalWindow API function, declare the API call in your global section, as follows:

```
Declare Function SetSysModalWindow Lib "User" (ByVal hwnd%) As Integer
```

At an appropriate place in your code, add the following:

```
Success% = SetSysModalWindow(hwnd)
```

Once this line is executed, your window will be the only window that can get focus until that window is destroyed.

Note: Because Visual Basic for Windows was not designed with system modal capabilities in mind, using a MsgBox, InputBox, or Form.Show of another form from a system modal window will not work correctly. If you want to show another window from a system modal form, use another Visual Basic for Windows form and call SetSysModalWindow for this second form also, so that it becomes the system modal window. When the second form is unloaded, the original system modal form will again become the

system modal window. Note that because the window(s) shown from a system modal window must also call SetSysModalWindow, and since MsgBox/InputBox windows cannot have associated code, you should not call the MsgBox or InputBox functions from a system modal window.

How to Access Windows Initialization Files Within Visual Basic

Article ID: Q75639

Summary:

There are several Microsoft Windows API functions that can manipulate information within a Windows initialization file. GetProfileInt, GetPrivateProfileInt, GetProfileString, and GetPrivateProfileString allow a Microsoft Visual Basic for Windows program to retrieve information from a Windows initialization file based on an application name and key name. WritePrivateProfileString and WriteProfileString are used to create/update items within Windows initialization files.

More Information:

Windows initialization files contain information that defines your Windows environment. Examples of Windows initialization files are WIN.INI and SYSTEM.INI, which are commonly found in the C:\WINDOWS subdirectory. Microsoft Windows and applications for Microsoft Windows can use the information stored in these files to configure themselves to meet your needs and preferences. For a description of initialization files, review the WIN.INI file that comes with Microsoft Windows.

An initialization file is composed of at least an application name and a key name. The contents of Windows initialization files have the following format:

```
[Application name]
keyname=value
```

There are four API function calls (GetProfileInt, GetPrivateProfileInt, GetProfileString, and GetPrivateProfileString) that you can use to retrieve information from these files. The particular function to call depends on whether you want to obtain string or numerical data.

The GetProfile family of API functions is used when you want to get information from the standard WIN.INI file that is used by Windows. The WIN.INI file should be part of your Windows subdirectory (C:\WINDOWS). The GetPrivateProfile family of API functions is used to retrieve information from any initialization file that you specify. The formal arguments accepted by these API functions are described farther below.

The WriteProfileString and WritePrivateProfileString functions write information to Windows initialization files. WriteProfileString is used to modify the Windows initialization file, WIN.INI. WritePrivateProfileString is used to modify any initialization file that you specify. These functions search the initialization file for the key name under the application name. If there is no match, the function adds to the user profile a new string entry containing the key name and the key value specified. If the key name is found, it will replace the key value with the new value specified.

To declare these API functions within your program, include the following Declare statements in the global module or the General Declarations section of a Visual Basic for Windows form:

```
Declare Function GetProfileInt% Lib "Kernel"(ByVal lpAppName$,
```

```
        ByVal lpKeyName$, ByVal nDefault%)

Declare Function GetProfileString% Lib "Kernel" (ByVal lpAppName$,
        ByVal lpKeyName$, ByVal lpDefault$, ByVal lpReturnedString$,
        ByVal nSize%)

Declare Function WriteProfileString% Lib "Kernel"(ByVal lpAppName$,
        ByVal lpKeyName$, ByVal lpString$)

Declare Function GetPrivateProfileInt% Lib "Kernel"
        (ByVal lpAppName$, ByVal lpKeyName$, ByVal nDefault%,
        ByVal lpFileName$)

Declare Function GetPrivateProfileString% Lib "Kernel"
        (ByVal lpAppName$, ByVal lpKeyName$, ByVal lpDefault$,
        ByVal lpReturnedString$, ByVal nSize%, ByVal lpFileName$)

Declare Function WritePrivateProfileString% Lib "Kernel"
        (ByVal lpAppName$, ByVal lpKeyName$, ByVal lpString$,
        ByVal lpFileName$)
```

Note: Each Declare statement must be on a single line.

The formal arguments to these functions are described as follows:

Argument	Description
lpAppName$	Name of a Windows application that appears in the initialization file.
lpKeyName$	Key name that appears in the initialization file.
nDefault$	Specifies the default value for the given key if the key cannot be found in the initialization file.
lpFileName$	Points to a string that names the initialization file. If lpFileName does not contain a path to the file, Windows searches for the file in the Windows directory.
lpDefault$	Specifies the default value for the given key if the key cannot be found in the initialization file.
lpReturnedString$	Specifies the buffer that receives the character string.
nSize%	Specifies the maximum number of characters (including the last null character) to be copied to the buffer.
lpString$	Specifies the string that contains the new key value.

Below are the steps necessary to create a Visual Basic for Windows sample program that uses GetPrivateProfileString to read from an initialization file that you create. The program, based on information in the initialization file you created, shells out to the Calculator program (CALC.EXE) that comes with Windows. The sample program

demonstrates how to use GetPrivateProfileString to get information from any initialization file.

1. Create an initialization file from a text editor (for example, you can use the Notepad program supplied with Windows) and save the file under the name of "NET.INI". Type in the following as the contents of the initialization file (NET.INI):

   ```
   [NetPaths]
   WordProcessor=C:\WINWORD\WINWORD.EXE
   Calculator=C:\WINDOWS\CALC.EXE
   ```

   Note: If CALC.EXE is not in the C:\WINDOWS subdirectory (as indicated after "Calculator=" above), replace C:\WINDOWS\CALC.EXE with the correct path.

2. Save the initialization file (NET.INI) to the root directory of your hard drive (such as C:\) and exit the text editor.

3. Start Visual Basic for Windows.

4. Create a form called Form1.

5. Create a push button called Command1.

6. Within the Global Declaration section of Form1, add the following Windows API function declarations. Note that the Declare statement below must appear on a single line.

   ```
   Declare Function GetPrivateProfileString% Lib "kernel"
     (ByVal lpAppName$, ByVal lpKeyName$,ByVal lpDefault$,
     ByVal lpReturnString$,ByVal nSize%, ByVal lpFileName$)
   ```

7. Within the (Command1) push button's click event add the following code:

```
Sub Command1_Click ()
   '* If an error occurs during SHELL statement then handle the error.
   On Error GoTo FileError

   '* Compare these to the NET.INI file that you created in step 1
   '* above.
   lpAppName$ = "NetPaths"
   lpKeyName$ = "Calculator"
   lpDefault$ = ""
   lpReturnString$ = Space$(128)
   Size% = Len(lpReturnString$)

   '* This is the path and name the NET.INI file.
   lpFileName$ = "c:\net.ini"

   '* This call will cause the path to CALC.EXE (that is,
   '* C:\WINDOWS\CALC.EXE) to be placed into lpReturnString$. The
   '* return value (assigned to Valid%) represents the number of
   '* characters read into lpReturnString$. Note that the
   '* following assignment must be placed on one line.
   Valid% = GetPrivateProfileString(lpAppName$, lpKeyName$,
```

```
                                    lpDefault$, lpReturnString$,
                                    Size%, lpFileName$)

   '* Discard the trailing spaces and null character.
   Path$ = Left$(lpReturnString$, Valid%)

   '* Try to run CALC.EXE. If unable to run, FileError is called.
   Succ% = Shell(Path$, 1)
   Exit Sub

FileError:
   MsgBox "Can't find file", 16, "Error lpReturnString"
   Resume Next

End Sub
```

How to Create and Use a Custom Cursor in Visual Basic; Win SDK

Article ID: Q76666

Summary:

Using a graphics editor, the Microsoft Windows Software Development Kit (SDK), and the Microsoft C compiler, you can create a dynamic-link library (DLL) containing mouse cursors that can be used in a Microsoft Visual Basic for Windows application. By making calls to the Windows API functions LoadLibrary, LoadCursor, SetClassWord, and GetFocus, you can display a custom cursor from within a Visual Basic for Windows application. Below are the steps necessary to a create a custom cursor and a Visual Basic for Windows application to use this custom cursor.

More Information:

Setting a custom cursor in a Visual Basic for Windows application requires a call to the Windows API function LoadLibrary to load the custom DLL containing the cursor resource(s). A call to LoadCursor is then required to load a single cursor contained in the DLL. The return value of the LoadCursor function is a handle to the custom cursor. This handle can be passed as an argument to the API function SetClassWord with the constant GCW_HCURSOR. SetClassWord also requires a window handle (hWnd) to the object (form or control) for which the cursor is to be set. The hWnd of a form is available via the hWnd run-time method. For example, the statement FWnd = Form1.hWnd will return the hWnd of Form1 to the variable FWnd. The hWnd of a control can be obtained by first using the SetFocus method on the control to give it the input focus and then calling the API function GetFocus. GetFocus returns the hWnd of the object with the current input focus.

A custom cursor always takes the place of the system cursor. The MousePointer property of a form or control to receive the custom cursor must be set to zero (system). Any other value for this property will result in the selected cursor being displayed, not the custom cursor.

Because the cursor is defined as part of a window class, any change to the window class will be reflected across any control or form that uses that class. For example, if the MousePointer property for two command buttons is zero (system) and a custom cursor is set for one of the command buttons, both of the command buttons will receive the custom cursor. To guarantee a custom cursor for each control requires that the cursor be set by calling SetClassWord in the MouseMove event procedure of the control.

Some controls, such as command buttons, do not contain a MouseMove event procedure. A custom mouse pointer for these types of controls can be set by initiating a timer event. Within the timer event, calls to the API functions GetCursorPos and WindowFromPoint can be made to determine if the mouse is over the control or not. If the WindowFromPoint API call returns the hWnd of the control, then the mouse is over the control. A call to SetClassWord can then be made to set the custom cursor for the control.

Below is an example of using the Windows SDK and C Compiler to create a DLL containing cursor resources. Further below are the steps necessary to create a Visual Basic for Windows program to use the cursor

resources.

If you do not have the Windows SDK but have a pre-compiled DLL containing cursor resources, skip to the steps below outlining how to create a Visual Basic application to use the custom cursor resources.

1. Using a graphics editor such as Microsoft Windows SDK Paint program, create two cursor images. Save the images separately as CURS1.CUR and CURS2.CUR, respectively.

2. Using any text editor, create a C source file containing the minimum amount of code for a Windows DLL. The source code must contain the functions LibEntry and WEP (Windows exit procedure). Below is an example of the C source file:

```
#include <windows.h>
int _far _pascal LibMain (HANDLE hInstance,
                          WORD wDataSeg,
                          WORD cbHeapSize,
                          LPSTR lpszCmdLine)
{
   return(1);
}

int _far _pascal WEP (int nParameter)
{
   return(1);
}
```

3. Save the file created in step 2 above as CURSORS.C.

4. Using any text editor, create a definition file (.DEF) for the DLL. Enter the following as the body of the .DEF file:

```
LIBRARY CURSORS

DESCRIPTION 'DLL containing cursor resources'

EXETYPE WINDOWS

STUB 'WINSTUB.EXE'

CODE MOVEABLE DISCARDABLE

DATA MOVEABLE SINGLE

HEAPSIZE 0

EXPORTS
   WEP   @1 RESIDENTNAME
```

5. Save the file created in step 4 above as CURSORS.DEF.

6. Using a text editor, create a resource file for the cursors created in step 1 above. Enter the following as the body of the .RC file:

```
Cursor1 CURSOR CURS1.CUR
```

```
Cursor2 CURSOR CURS2.CUR
```

7. Save the file created in step 6 above as CURSORS.RC.

8. Compile CURSORS.C from the MS-DOS command line as follows:

```
CL /AMw /c /Gsw /Os /W2 CURSORS.C
```

9. Link the program from the MS-DOS command line as follows (enter the following two lines on a single line):

```
LINK /NOE /NOD cursors.obj +
           LIBENTRY.OBJ,,,MDLLCEW.LIB+LIBW.LIB,CURSORS.DEF;
```

   This will create the file CURSORS.EXE.

10. Add the cursor resources created in step 1 above to the .EXE file created in step 9 above by invoking the Microsoft Resource Compiler (RC.EXE) from the MS-DOS command line as follows:

```
RC CURSORS.RC
```

11. Rename CURSORS.EXE to CURSORS.DLL from the MS-DOS command line as follows:

```
REN CURSORS.EXE CURSORS.DLL
```

Below are the steps necessary to create a Visual Basic for Windows application that uses the cursor resources created in the steps above.

Important

When running the Visual Basic for Windows program created by following the steps below, it is important to terminate the application from the system menu, NOT the Run End option from the file menu. When Run End is chosen from the file menu, the unload event procedure is not executed. Therefore, the system cursor is not restored and the custom cursor will remain present at design time. Using Visual Basic version 1.0 for Windows, avoid terminating the program from the Program Manager (PROGMAN.EXE) task list. The unload event procedure is also not called when a program is terminated from the task list in Visual Basic version 1.0 for Windows.

1. Start Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 will be created by default.

2. Put a picture control (Picture1) on Form1.

3. Put a command button (Command1) on Form1.

4. Put a timer control (Timer1) on Form1.

5. Enter the following code in the Global Module:

```
Type PointType
    x As Integer
```

```
    y As Integer
End Type
```

6. Enter the following code in the General Declaration section of Form1:

```
DefInt A-Z
' Each of the following Declare statements must appear on one line.
Declare Function LoadLibrary Lib "kernel" (ByVal LibName$)
Declare Function LoadCursor Lib "user"    (ByVal hInstance, ByVal
                                           CursorName$)
Declare Function SetClassWord Lib "user" (ByVal hWnd, ByVal
                                          nIndex, ByVal NewVal)
Declare Function DestroyCursor Lib "user" (ByVal Handle)
Declare Function GetFocus Lib "user" ()
Declare Function PutFocus Lib "user" Alias "SetFocus" (ByVal hWnd)
Declare Sub GetCursorPos Lib "user" (p As PointType)
Declare Function WindowFromPoint Lib "user" (ByVal y, ByVal x)
Const GCW_HCURSOR = (-12)
Dim SysCursHandle
Dim Curs1Handle
Dim Curs2Handle
Dim Pic1hWnd
Dim Command1hWnd
Dim p As PointType
```

7. Enter the following code in the Form_Load event procedure of Form1:

```
Sub Form_Load ()
    Form1.Show
    DLLInstance = LoadLibrary("CURSORS.DLL")
    Curs1Handle = LoadCursor(DLLInstance, "Cursor1")
    Curs2Handle = LoadCursor(DLLInstance, "Cursor2")
    SysCursHandle=SetClassWord(Form1.hWnd,GCW_HCURSOR,Curs2Handle)

    ' Get the current control with the input focus.
    CurrHwnd = GetFocus()

    ' Get the Window handle of Picture1.
    Picture1.SetFocus
    Pic1hWnd = Picuture1.GetFocus()

    ' Get the Window handle of Command1.
    Command1.SetFocus
    Command1hWnd = GetFocus()

    ' Restore the focus to the control with the input focus.
    r = PutFocus(CurrHwnd)
    timer1.interval = 1   ' One millisecond.
    timer1.enabled = -1
End Sub
```

8. Enter the following code in the Form_Unload event procedure of Form1:

```
  Sub Form_Unload (Cancel As Integer)
```

```
    ' Restore the custom cursors to the system cursor:
      LastCursor =SetClassWord(Form1.hWnd, GCW_HCURSOR, SysCursHandle)
      LastCursor = SetClassWord(Pic1hWnd, GCW_HCURSOR, SysCursHandle)
      LastCursor=SetClassWord(Command1hWnd, GCW_HCURSOR,SysCursHandle)
    ' Delete the cursor resources from memory:
      Success = DestroyCursor(Curs1Handle)
      Success = DestroyCursor(Curs2Handle)
  End Sub
```

9. Enter the following code in the Timer1_Timer event procedure of Timer1:

```
  Sub Timer1_Timer ()

      ' Get the current (absolute) cursor position.
      Call GetCursorPos(p)

      ' Find out which control the midpoint of the cursor is over.
      ' The cursor is 32 x 32 pixels square. Change the class word
      ' of the control to the appropriate cursor.
      Select Case WindowFromPoint(p.y + 16, p.x + 16)

       Case Form1.hWnd
       ' Each of the following statements must appear on one line.
           LastCursor = SetClassWord(Form1.hWnd, GCW_HCURSOR,
                                     Curs2Handle)
           LastCursor = SetClassWord(Command1hWnd, GCW_HCURSOR,
                                     Curs2Handle)
           LastCursor = SetClassWord(Pic1hWnd, GCW_HCURSOR,
                                     Curs2Handle)

        Case Command1hWnd

           LastCursor = SetClassWord(Form1.hWnd, GCW_HCURSOR,
                                     Curs1Handle)
           LastCursor = SetClassWord(Command1hWnd, GCW_HCURSOR,
                                     Curs1Handle)

        Case Pic1hWnd

           LastCursor = SetClassWord(Form1.hWnd, GCW_HCURSOR,
                                       Curs1Handle)
           LastCursor = SetClassWord(Pic1hWnd%, GCW_HCURSOR,
                                       Curs1Handle)
      End Select
  End Sub
```

Run the program. The form should receive the "Cursor2" cursor and the controls Command1 and Picture1 should receive the "Cursor1" cursor as the mouse cursor is moved about the form.

How to Play a Waveform (.WAV) Sound File in Visual Basic

Article ID: Q86281

Summary:

You can play a waveform (.WAV) sound file from Microsoft Visual Basic for Windows by calling the sndPlaySound API function from the MMSYSTEM.DLL file. In order to be able to call the sndPlaySound API function, you must be using either Microsoft Windows, version 3.1 or the Microsoft Multimedia Extensions for Windows, version 3.0. The following information discusses the sndPlaySound parameters, and includes an example of how to use this function from Visual Basic for Windows.

More Information:

To use the sndPlaySound API from within a Visual Basic for Windows application, you must Declare the sndPlaySound function in either the global module or from within the Declarations section of your Code window. Declare the function as follows:

```
Declare Function sndPlaySound Lib "MMSTSTEM.DLL" (ByVal lpszSoundName$
                                                  ByVal wFlags%) As Integer
```

Note: The above Declare statement must be written on just one line.

The parameters listed above are explained as follows:

Parameters
---

lpszSoundName$

Specifies the name of the sound to play. The function first searches the [sounds] section of the WIN.INI file for an entry with the specified name, and plays the associated waveform sound file. If no entry by this name exists, then it assumes the specified name is the name of a waveform sound file. If this parameter is NULL, any currently playing sound is stopped.

wFlags%

Specifies options for playing the sound using one or more of the following flags:

SND_SYNC
The sound is played synchronously and the function does not return until the sound ends.

SND_ASYNC
The sound is played asynchronously and the function returns immediately after beginning the sound.

SND_NODEFAULT
If the sound cannot be found, the function returns silently without playing the default sound.

SND_LOOP

The sound will continue to play repeatedly until sndPlaySound is called again with the lpszSoundName$ parameter set to null. You must also specify the SND_ASYNC flag to loop sounds.

SND_NOSTOP
If a sound is currently playing, the function will immediately return False without playing the requested sound.

The sndPlaySound function returns True (-1) if the sound is played, otherwise it returns False (0).

The following code example illustrates how to use the sndPlaySound API function to play a waveform (.WAV) sound file.

Add the following code to the global module or general Declarations section of your form:

```
' The following Declare statement must appear on one line.
Declare Function sndPlaySound Lib "MMSYSTEM.DLL" (ByVal
             lpszSoundName$, ByVal wFlags%) As Integer

Global Const SND_SYNC      = &H0000
Global Const SND_ASYN      = &H0001
Global Const SND_NODEFAULT = &H0002
Global Const SND_LOOP      = &H0008
Global Const SND_NOSTOP    = &H0010
```

Add the following line of code to the appropriate function or subroutine in your application:

```
SoundName$ = "c:\windows\tada.wav"
wFlags% = SND_ASYNC And SND_NODEFAULT
x% = sndPlaySound(SoundName$,wFlags%)
```

Note that if a large waveform (.WAV) sound file is specified and the above call fails to play the file in its entirety, you will need to adjust the settings on the appropriate sound driver.

Visual Basic for Windows Reference Materials

Article ID: Q12345

Summary:

The following is a list of Microsoft Windows API references:

"Microsoft Windows 3.1 Software Development Kit: Reference Volume 1"

"Microsoft Windows 3.1 Software Development Kit: Reference Volume 2

"Microsoft Windows Programmer's Reference Book and Online Resource"
(Add-on kit number 1-55615-413-5)

"Microsoft Windows Programmer's Reference"

"Microsoft Multimedia Development Kit: Programmer's Reference"
version 1.0

"Programming Windows: the Microsoft Guide to Writing Applications
for Windows 3" by Charles Petzold, Microsoft Press, 1990

WINSDK.HLP file shipped with the Professional Edition of Microsoft
Visual Basic for Windows, version 3.0

F5 in Run Mode with Focus on Main Menu Bar Acts as CTRL+BREAK

Article ID: Q74348

Summary:

A Microsoft Visual Basic for Windows program will break at run time under the following simultaneous conditions:

1. You run the program in the Visual Basic for Windows development environment.

2. The Visual Basic for Windows menu bar has the focus.

3. You press the F5 key.

The program will break when the F5 key is pressed and the Immediate Window will get the focus. This is not a problem with Visual Basic for Windows, but rather a design feature.

This information only applies to an application run in the Visual Basic for Windows development environment, not as an .EXE program.

More Information:

The F5 key acts as the shortcut key for the Visual Basic for Windows Run menu. Because Start, Continue, and Break all share the same menu item under the Run menu, F5 acts differently depending upon the state of execution of a program. It acts as the Run key in the Visual Basic version 1.0 for Windows environment. It also serves as the Break key once the application is running and the focus is on the Visual Basic for Windows menu bar. After execution has been "broken" with the Break key, the F5 key serves as the Continue key.

To demonstrate the different modes of the F5 key, do the following:

1. Run Visual Basic for Windows.

2. From the File menu, select New Project (press ALT, F, N).

3. Press the F5 key to run the program.

4. Using the mouse, click on the Visual Basic for Windows menu bar.

5. Press the F5 key to break the program. The Immediate window will be given the focus after you press the F5 key.

6. Press the F5 key again to continue execution of the program.

Visual Basic SendKeys Statement Is Case Sensitive

Article ID: Q81466

Summary:

The SendKeys statement in Microsoft Visual Basic for Windows is case sensitive with regards to the keystrokes sent. Sending an uppercase letter may be interpreted by the receiving application differently than the lowercase version of a letter.

More Information:

The following line of code sends an ALT+F key combination to the application that currently has the focus:

```
SendKeys "%(F)"
```

Note that this is different than ALT+f:

```
SendKeys "%(f)"
```

This can be a problem because some applications distinguish between an uppercase F and lower case f when sent by the SendKeys statement.

For example, Microsoft Word versions 1.0b and earlier for Windows (WINWORD.EXE) do not distinguish the difference. However, Microsoft Word version 2.0 for Windows does distinguish the lowercase f sent by SendKeys.

When SendKeys (from Visual Basic for Windows) sends the ALT+F key combination, WINWORD.EXE version 2.0 interprets the keystroke as ALT+Shift+f, at which Word for Windows will simply beep. However, SendKeys using ALT+f will correctly activate the File menu.

No New Timer Events During Visual Basic Timer Event Processing

Article ID: Q78599

Summary:

Timer controls can be used to automatically generate an event at predefined intervals. This interval is specified in milliseconds, and can range from 0 to 65535 inclusive.

Timer event processing will not be interrupted by new timer events. This is because of the way that Windows notifies an application that a timer event has occurred. Instead of interrupting the application, Windows places a WM_TIMER message in its message queue. If there is already a WM_TIMER message in the queue from the same timer, the new message will be consolidated with the old one.

After the application has completed processing the current timer event, it checks its message queue for any new messages. This queue may have new WM_TIMER messages to process. There is no way to tell if any WM_TIMER messages have been consolidated.

Scope of Line Labels/Numbers in Visual Basic for Windows

Article ID: Q78335

Summary:

Line labels (and line numbers) do not follow the same scoping rules as variables and constants in Visual Basic for Windows. Line labels must be unique within each module and form. However, you can only transfer control to a line label or line number within the current Sub or Function.

More Information:

When you attempt to define the same line label twice within a module or form, you receive the error message "Duplicate label". This message means that the label is already defined in another procedure within the current module.

When you use a GOTO or GOSUB statement that names a line label defined in another procedure, you receive the error message "Label not defined." This message means that the label is not defined in the current Sub or Function.

For more information about line labels, see the description of the GOTO and GOSUB statements in the "Microsoft Visual Basic: Language Reference" or in the Visual Basic for Windows online Help system.

Sending Keystrokes from Visual Basic to an MS-DOS Application

Article ID: Q77394

Summary:

The "Microsoft Visual Basic: Language Reference" version 1.0 manual states that the SendKeys function cannot be used to send keystrokes to a non-Windows application. Listed below is a method that can be used to work around this limitation.

More Information:

The Microsoft Visual Basic for Windows SendKeys function can send keystrokes to the currently active window (as if the keystrokes had been typed at the keyboard). Although it is not possible to send keystrokes to a non-Windows application with SendKeys directly, you can place text on the Clipboard and use SendKeys to paste that text into an MS-DOS application that is running in a window (or minimized as an icon.)

To run an MS-DOS application in a window, you MUST be running in Windows 386 enhanced mode. You must also make sure that the MS-DOS application's PIF file has been set to display the application in a window rather than full screen. Use the Windows PIF Editor to make this modification, if necessary.

An example of sending keystrokes to an MS-DOS session running in a window is given below:

1. Start a MS-DOS session (running in a window).

2. Start Visual Basic for Windows.

3. Enter the following into the general declarations section of the form:

   ```
   Dim progname As String
   ```

4. Draw two labels on the form. Change the first label's caption to "Dos App Title." Change the second label's caption to "Keys to send."

5. Draw two text boxes on the form (next to each of the previously drawn labels). Delete the default contents of these text boxes. These controls will be used to allow the user to enter the MS-DOS application window title and the keystrokes to send to it. Change the Name property of these text boxes to "DosTitle" and "DosKeys," respectively.

6. Draw a command button on the form and change its caption to "Send keys."

7. Attach the following code to the command button click procedure:

   ```
   progname = "Microsoft Visual Basic"
   clipboard.Clear
   clipboard.SetText DosKeys.Text + Chr$(13)  ' Append a <CR>.
   AppActivate DosTitle.Text
   ```

```
SendKeys "% ep", 1
AppActivate progname
```

If the text that you send is the DIR command or another command that takes time, the AppActivate call immediately following the SendKeys call will interrupt the processing. The AppActivate call should be placed in a timer with the appropriate interval set and the timer enabled in the command_click procedure. The timer should be disabled before exiting the timer.

8. Run the program.

9. Enter the window title of the MS-DOS application into the DosTitle text box. The default window title for an MS-DOS session is "DOS." If you would like to change the window title of an MS-DOS application, you should use the PIF Editor.

10. Enter the keystrokes to send into the DosKeys text box (for example, "DIR").

11. Click on the Send Keys command button. The keystrokes will be sent to the Clipboard and then pasted into the MS-DOS window.

If this technique is used in a compiled Visual Basic for Windows program, you should change the progname assignment from "Microsoft Visual Basic" to the executable file name. Also, if you would like to see the text being placed onto the Clipboard, you can open the Windows Clipboard viewer.

Task List Switch to VB Application Fails After ALT+F4 Close

Article ID: Q81469

Summary:

Selecting the Close command from the Control menu (ALT+F4) to quit a Visual Basic for Windows application will not necessarily unload any other forms that have been loaded. If other forms have been loaded but are not visible, the application may still be running under Windows. If this is the case, the Windows Task List will still contain the name of the application. Attempting to switch to the application from the Windows Task List will be unsuccessful.

If you want the application to terminate as a result of unloading a particular form, place an End statement in the Form_Unload event procedure for the form, or use the Unload statement to unload all forms that are loaded. This will cause all forms (visible and invisible) to be unloaded, and the application to terminate.

More Information:

Even if the form that is closed is the designated startup form in your application, it will not automatically unload previously loaded forms. Therefore, the application can in fact still be running and appear in the Windows Task List. You can terminate the application by selecting the End Task button in the Windows Task List, but you will not be able to switch to the task.

Below are the steps necessary to cause an application to terminate when a particular form is closed from the Control menu (ALT+F4).

With the application loaded in VB.EXE (the Visual Basic for Windows development environment), do the following:

1. Double-click on the form to open the Code window.

2. Add an End statement to the Form_Unload event procedure for the form. For example:

```
Sub Form_Unload (Cancel As Integer)

     ' Your code goes here.
     '


     End  ' This will unload all forms and terminate the
          ' application.
End Sub
```

Adding an End statement to the Unload event procedure of a form will not cause the Unload event procedures for the other forms to be called. To cause the Unload event procedures for the other forms to be called, use the Unload statement to explicitly unload each form.

Example of Sharing a Form Between Projects in VB for Windows

Article ID: Q81222

Summary:

Microsoft Visual Basic for Windows allows you to share forms between projects. When you make a change to a shared form in one project, that change will be automatically updated in the other projects that share the form.

A workaround is also available if you want to change a shared form but do not want to update the form in other projects.

Further below is an example of how to use this shared form feature in Visual Basic for Windows, and an example of how to change a shared form without updating it in shared projects.

More Information:

Below are two examples: the first shows how to update shared forms, and the second demonstrates how to change a shared form without having those changes affect the same form in other projects.

Example 1
---

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Add a couple text boxes and command buttons to Form1 by double-clicking on the appropriate tools in the toolbox and placing the controls at certain locations on the form. From the Properties Bar, change the FormName property of Form1 to Test1.

3. From the File menu, choose Save Project As. Save Test1 as TEST1.FRM and save the project as TEST1.MAK.

4. Start a new project by choosing New Project from the File menu.

5. From the File menu, choose Add File, and select TEST1.FRM.

6. Once TEST1.FRM is loaded into the project, delete the command buttons, and replace them with picture boxes.

7. From the File menu, choose Save Project As. Save the project as TEST2.MAK, and save TEST1.FRM with the same name.

8. From the File menu, choose Open Project. In the Files box, select TEST1.MAK.

Notice that the form has been updated to include picture boxes and the command buttons were deleted.

Example 2

(Note that the following steps are very similar to the example above,

but with a change in step 5.)

This example demonstrates how to share forms between projects, but with the forms being designed differently.

1. Run Visual Basic for Windows, or from the File menu, choose New Project (press ALT, F, N) if Visual Basic for Windows is already running. Form1 is created by default.

2. Add a couple text boxes and command buttons to Form1 by double-clicking on the appropriate tools in the toolbox and placing the controls at certain locations on the form. From the Properties Bar, change the FormName property of Form1 to Test3.

3. From the File menu, choose Save Project As. Save Test3 as TEST3.FRM and save the project as TEST3.MAK.

4. From the File menu, choose New Project.

5. From the File menu, choose Add File. In the Files box, select TEST3.FRM. Once the file is loaded, delete the command buttons and replace them with picture boxes.

6. From the File menu, choose Save File As, and save the form as TEST4.FRM.

7. From the File menu, choose Save Project As, and save the project as TEST4.MAK.

8. From the File menu, choose Open Project. In the Files box, select TEST3.MAK.

Notice that the form's controls have NOT been updated with picture boxes.

"Property or Control Not Found" Using Form/Control Data Type

Article ID: Q84383

Summary:

You do not need to prefix a control name with the parent form name when you are accessing the property of a control from a Sub or Function to which the control is passed as a parameter. If you use the syntax

```
form.control.property
```

to access the property of the control, you will get a "Property or Control not found" error.

More Information:

The full syntax to access a property of a control on a form is as follows:

```
form.control.property
```

If the control whose property you are accessing is on the form where the code resides, you do not need to prefix the control name with the form name. For example, if command button Command1 is on Form1 and you want to set its Enabled property to False (0) in the event procedure Command1_Click, you can use the following:

```
Command1.Enabled = 0
```

You can use the same syntax if the statement is in the general Declarations section of Form1. However, if you want to access the Enabled property of Command1 on a form other than its parent form, or from a Sub or Function in a module, you need to use the full syntax (with the form name).

The property of the control can also be accessed in a module by using the full syntax. For example, to disable Command1 (which is on Form1) in MODULE1.BAS, add the following:

```
Sub AccessProperty
    Form1.Command1.Enabled = 0
End Sub
```

However, if you are passing the control as an argument to a Sub or Function procedure in a general module, you do not need to use the full syntax. For example

```
Sub AccessProperty (ThisForm as Form, ThisControl as Control)
    ThisForm.ThisControl.Enabled = 0
End Sub
```

will give you a "Property or Control 'ThisControl' not found" error. You only need to pass the control name as an argument to the procedure. For example:

```
Sub AccessProperty (ThisControl as Control)
    ThisControl.Enabled = 0
```

```
End Sub
```

Avoid Could not execute: SETUP1.EXE 2" Error, Use COMPRESS-r
Article ID: Q93426
Summary:

Files used with the Setup Kit must be either uncompressed or compressed with the command COMPRESS -r <filename>. The following error can occur if you use a method other than COMPRESS -r to create a file with an underscore as the last character:

   Error - Could not execute: SETUP1.EXE 2

However, VER.DLL must be named VER.DL_ on the setup disk and must not be compressed.

More Information:

The filename listed in the error message above can be different than SETUP1.EXE if you customized the Setup Kit.

The following two commands both create a file named SETUP1.EX_, but they are not equivalent:

```
COMPRESS -r SETUP1.EXE              (correct)
COMPRESS SETUP1.EXE SETUP1.EX_      (incorrect)
```

The COMPRESS.EXE option -r compresses a file, replaces the last character of the filename with an underscore "_", and stores the replaced character in the compressed file. When the Setup Kit uses VER.DLL to decompress a file, VER.DLL reads the character from the file and restores the file to its original name.

If you create a file with an underscore as the last character without using COMPRESS -r, VER.DLL renames the file by removing the underscore. For example, SETUP1.EX_ becomes SETUP1.EX.