Dynamic Data Exchange Issues and Examples

Dynamic data exchange (DDE) is a mechanism supported by the Microsoft Windows operating environment that enables two applications to "talk" to each other by continuously and automatically exchanging data. DDE automates the manual cutting and pasting of data between applications, providing a faster vehicle for updating information.

The articles below contain some examples of communicating with several popular applications, including a Visual Basic application. Some design issues are also covered.

Example of Client/Server DDE Between Visual Basic Applications

Article Number: Q74861

Summary:

This article outlines the steps necessary to initiate dynamic data exchange (DDE) between a Microsoft Visual Basic client application and a Visual Basic server application.

This article demonstrates how to:

1. Create a Visual Basic application that will function as a server.

2. Create a Visual Basic application that will function as a client.

3. Initiate a cold DDE link (information updated upon request from the client) between the client application and the server application.

4. Use LinkRequest to update information in the client application from information in the server application.

5. Initiate a hot DDE link (information updated automatically from server to client) between the client application and the server application.

6. Use LinkPoke to send information from the client application to the server application.

7. Change the LinkMode property between hot and cold.

More Information:

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

Example

The steps below show how to establish a DDE conversation between two Visual Basic applications.

First, create the server application in Visual Basic:

1. Start Visual Basic, and Form1 will be created by default.

2. Change the Caption property of Form1 to "Server".

3. Put a Text Box (Text1) on Form1.

4. Save the form and project with the name SERVER.

5. From the File menu, choose Make EXE File. In the Make EXE File dialog box, choose the OK button to accept SERVER.EXE as the name of the EXE file.

Second, create the client application in Visual Basic:

1. From the File menu, choose New Project. Form1 will be created by default.

2. Change the Caption property of Form1 to "Client".

3. Create the following controls with the following properties on Form1:

Default Name	Caption	CtlName
Text1	(N/A)	Text1
Option1	Cold Link	ColdLink
Option2	Hot Link	HotLink
Command1	Poke	Poke
Command2	Request	Request

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

```
Const TRUE = -1
Const FALSE = 0
Const HOT = 1
Const COLD = 2
Const NONE = 0
```

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

```
Sub Form_Load ()
    z% = Shell("C:\VB\SERVER", 1)
      'Causes Windows to finish processing the Shell command.
    z% = DoEvents()
      'Clears DDE link.
    Text1.LinkMode = NONE
      'Sets up link with VB server.
    Text1.LinkTopic = "Server|Form1"
      'Set link to text box server
    Text1.LinkItem = "Text1"
      'Establish a cold DDE link.
    Text1.LinkMode = COLD
      'Sets appropriate button.
    ColdLink.Value = TRUE
End Sub
```

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

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

7. Add the following code to the Clink event procedure of HotLink:

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

```
End Sub
```

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

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

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

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

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

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

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

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

4. Run the application (from Windows if an .EXE file, or from the Run menu if from the VB.EXE environment.)

5. Form1 of the client application will load and the server application will automatically start.

You can now experiment with DDE between Visual Basic applications:

1. Try typing some text into the server's text box and then click the Update button. The text appears in the client's text box.

2. Click the Hot Link button, then type some more text into the server's text box. The text is automatically updated in the client's text box.

3. Type some text in the client's text box and click the Poke button. The text is sent to the server's text box.

You can also establish DDE between applications at design time, as described on page 356 of the "Microsoft Visual Basic: Programmer's Guide" version 1.0 manual.

DDE Example Between Visual Basic and Word for Windows

Article Number: 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:

1. Prepare a Word for Windows document for active DDE.

2. Initiate a cold DDE link (information updated upon request from the client) between the Visual Basic application (the client) and the document loaded into Word for Windows (the server).

3. Use LinkRequest to update information in the Visual Basic client based on information contained in the Word for Windows server.

4. Initiate a hot DDE link (information updated automatically from server to client) between the Visual Basic client and the Word for Windows server.

5. Use LinkPoke to send information from the Visual Basic client to the Word for Windows server.

6. Change the LinkMode property between hot and cold.

More Information:

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

Example

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

First, create the server document in Word for Windows:

1. Start Word for Windows. "Document1" will be created by default.

2. From the Window menu, choose Arrange All. This unmaximizes the document if it was maximized. Note that the title at the top of the WINWORD.EXE main title bar is now "Microsoft Word" and NOT "Microsoft Word - Document1".

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

4. From the Insert menu, choose Bookmark. Under Bookmark Name, type "DDE_Link" (without quotation marks). Press ENTER. This sets a bookmark for the entire document. This bookmark will function as the LinkItem in the DDE conversation.

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

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

Second, create the client application in Visual Basic:

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

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

Default Name	Caption	CtlName
Text1	(N/A)	Text1
Option1	Cold Link	ColdLink
Option2	Hot Link	HotLink
Command1	Poke	Poke
Command2	Request	Request

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

```
Const TRUE = -1
Const FALSE = 0
Const HOT = 1
Const COLD = 2
Const NONE = 0
```

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

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

    z% = Shell("C:\WinWord\WinWord C:\WinWord\Server.Doc", 1)
      'Process Windows events. This ensures that WinWord will be executed before any attempt is
                made to perform DDE with it.
    z% = DoEvents ()
      'Clears DDE link if it already exists.
    Text1.LinkMode = NONE
      'Sets up link with WINWORD.EXE.
    Text1.LinkTopic = "WinWord|\WinWord\Server"
      'Set link to bookmark on document.
    Text1.LinkItem = "DDE_Link"
      'Establish a cold DDE link.
    Text1.LinkMode = COLD
    ColdLink.Value = TRUE
End Sub
```

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

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

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

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

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

```
Sub Request_Click ()
        'With a cold DDE link this button will be visible, and clicking this button will request an update
                of information from the server application to the client 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 clicking this button will poke information
                from the client application into the server application.
    Text1.LinkPoke
End Sub
```

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

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

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

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

4. Run the application (from Windows if an .EXE file, or from the Run menu if from the Visual Basic environment.) Form1 of the Visual Basic client application will be loaded, and Word for Windows will automatically be started with the document SERVER.DOC loaded.

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

You can now experiment with DDE between Visual Basic and Word for Windows:

1. Try typing some text in the document in Word for Windows, and then click the Request button. The text appears in the text box.

2. Click the Hot Link button, then type some more text in 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 and click the Poke button. The text is sent to the document in Word for Windows.

Note: If you delete the total contents of the bookmark in the Word for Windows document, the bookmark will also be deleted. Any further attempts to perform DDE with this WINWORD.EXE session after the bookmark has been deleted will produce the following error message:

Foreign application 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.

You can also establish DDE between applications at design time, as described on page 356 of the "Microsoft Visual Basic: Programmer's Guide" version 1.0 manual.

LinkTimeOut of -1 Waits Only 6553.5 Seconds Before Timing Out

Article Number: Q77243

Summary:

Contrary to the documentation and online Help for Microsoft Visual Basic, setting the LinkTimeOut property of a control to -1 will not cause the control to wait forever for a DDE operation to complete. Setting the LinkTimeOut property to -1 will cause the control to wait for 65535 intervals of 1/10 second, for a total of approximately 1 hour and 49 minutes.

More Information:

To work around this problem, you can trap the DDE time-out error using the On Error statement in Visual Basic. If the error was "Timeout while waiting for DDE response" you can retry the DDE operation until it succeeds. The following is a code example:

```
Sub DDE_Retry_Forever (Source as Control, command$)
    On Local Error Goto Handler

    Source.LinkExecute command$
    Exit Sub

Handler:
    If Err = 286 Then
        Resume
    Else
        Error Err
    End If
End Sub
```

DDE from Visual Basic to Excel for Windows

Article Number: Q75089

Summary:

This article describes how to initiate a dynamic data exchange (DDE) conversation between a Visual Basic client application and a Microsoft Excel server application.

This article demonstrates how to:

1. Prepare a Microsoft Excel for Windows document for active DDE.

2. Initiate a cold DDE link (information updated upon request from the client) between Visual Basic (the client) and Excel (the server).

3. Use LinkRequest to update information in Visual Basic (the client) based on information contained in Excel (the server).

4. Initiate a hot DDE link (information updated automatically from server to client) between Visual Basic (the client) and Excel (the server).

5. Use LinkPoke to send information from Visual Basic (the client) to Excel (the server).

6. Change the LinkMode property between hot and cold.

More Information:

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

The steps below serve as a example of how to establish a DDE conversation between Visual Basic and Excel for Windows.

First, create the server spreadsheet in Excel:

1. Start Excel, and a document titled "SHEET1" will be created by default.

2. From the File menu, choose Save As, and save the document with the name SERVER.XLS.

3. Exit Excel. For this example to function properly, Excel must not be loaded and running.

Second, create the client application in Visual Basic:

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

1. Start Visual Basic (VB.EXE), and Form1 will be created by default.

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

Default Name	Caption	CtlName
Text1	(N/A)	Text1
Option1	Cold Link	ColdLink
Option2	Hot Link	HotLink

Command1	Poke	Poke
Command2	Request	Request

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

```
Const TRUE = -1
Const FALSE = 0
Const HOT = 1
Const COLD = 2
Const NONE = 0
```

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

```
Sub Form_Load ()
        'This procedure will start Excel and load SERVER.XLS, the spreadsheet that was created
                earlier.
    z% = Shell("C:\EXCEL\EXCEL c:\excel\SERVER.XLS", 1)
      'Process Windows events. This ensures that Excel will be executed before any attempt is
                made to perform DDE.
    z% = DoEvents()
      'Clears DDE link if it already exists.
    Text1.LinkMode = NONE
      'Sets up link with Excel.
    Text1.LinkTopic = "Excel|c:\excel\server.xls"
      'Set link to first cell on spreadsheet.
    Text1.LinkItem = "R1C1"
      'Establish a cold DDE link.
    Text1.LinkMode = COLD
    ColdLink.Value = TRUE
End Sub
```

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

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

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

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

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

```
Sub Request_Click ()
        'With a cold DDE link this button will be visible and when selected it will request an update of
                information from the server application to the client 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 client application to the server application.
    Text1.LinkPoke
End Sub
```

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

1. From the Visual Basic File menu, save the Form and Project with the name CLIENT.

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

3. Exit Visual Basic.

4. Run the application (from Windows if an .EXE file or from the Run menu if from the Visual Basic environment).

5. Form1 of the client application will be loaded and Excel will automatically be started with the document SERVER.XLS loaded.

6. Make sure that the main title bar in Excel reads "Microsoft Excel," NOT "Microsoft Excel - SERVER.XLS." If the title bar is incorrect, then from the Window menu choose Arrange All.

You can now experiment with DDE between Visual Basic and Excel:

1. Try typing some text in R1C1 in the spreadsheet and then choose the Request button. The text appears in the text box.

2. Choose the Hot Link button and then type some more text in R1C1 of the spreadsheet. The text is automatically updated in the Visual Basic text box.

3. Type some text in the text box in the Visual Basic application and choose the Poke button. The text is sent to R1C1 in the Excel spreadsheet.

Note that if you have "Ignore Remote Requests" checked in the "Options" - "Work space" menu, you will not be able to establish DDE from Visual Basic. Make sure that "Ignore Remote Requests" is NOT checked.

You can also establish DDE between applications at design time. This is covered on page 356 in the "Microsoft Visual Basic: Programmer's Guide" version 1.0 manual.

Cannot Make DDE Link to Object as Child of Another Object

Article Number: Q75091

Summary:

When performing dynamic data exchange (DDE) from a Visual Basic client application to any Windows server application (including Visual Basic server applications), you cannot make a DDE link to an object that is not a direct child of a form in the Visual Basic application. For example, with Microsoft Excel for Windows, you cannot establish a DDE link from a text box in a Visual Basic application if that text box is contained within a picture box on Form1. The text box must be on Form1 and not a child of any other object on that form.

More Information:

To work around this problem, create an additional object on the form and set that object's Visible property to False. You can then establish a DDE link with this hidden object.

Establishing a DDE link with a hidden object will issue the Change event for this object. You can attach code to this event that will update the information sent via the DDE link to the appropriate control, which may or may not be a direct child of the form.

The following example demonstrates this workaround:

First, create the server application in Visual Basic:

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

2. Put a text box (Text1) on Form1.

3. Save the Form and Project with the name SERVER.

4. From the File menu, choose Make EXE File, make an .EXE file, and call it SERVER.EXE.

Next, create the client application in Visual Basic:

1. From the File menu, choose New Project. Form1 will be created by default.

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

3. Put a text box (Text1) in the picture box (Picture1) as follows:

   a. Click ONCE on the text box icon in the ToolBox.

   b. Move the mouse pointer on top of Picture1 (note the mouse pointer is a cross).

   c. Press the left mouse button and drag the mouse pointer to size the text box.

   d. Release the left mouse button. Text1 is now a child of Picture1.

4. Put a text box (Text2) on Form1.

5. Set the Visible property of Text2 to False.

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

```
Const NONE = 0
Const HOT = 1
```

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

```
Sub Form_Load ()
        'This procedure will start the Visual Basic server application that was created earlier. Show
                Form1
    Form1.Show
    z% = Shell("C:\VB\SERVER, 1)
      'Causes Windows to process the Shell.
    z% = DoEvents()
      'Clears DDE Link if already there.
    Text2.LinkMode = NONE
      'Sets up link with VB server.
    Text2.LinkTopic = "Server|Form1"
      'Sets link to Text Box on server.
    Text2.LinkItem = "Text1"
      'Establish a hot DDE link.
    Text2.LinkMode = HOT
End Sub
```

8. Add the following code to the Change event procedure of Text1:

```
Sub Text2_Change ()
        'This procedure will cause the information in the text field of Text1 to be updated whenever
                the text property of Text2 changes.
    Text1.Text = Text2.Text
End Sub
```

You can now run the Visual Basic client application:

1. From the Run menu, choose Start.

2. Form1 of the client application will be loaded and the Visual Basic server application will automatically be started.

3. Any information you now type in the text box of the server application will be automatically updated to the invisible text box (on Form1) in the client application and then automatically transferred to the Visible text box (on Picture1 on Form1) in the client application.

DDE Between Visual Basic and Q+E for Windows

Article Number: Q75090

Summary:

This article describes how to initiate a dynamic data exchange (DDE) conversation between a Visual Basic client application and a Pioneer Software Q+E for Windows server 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 cold DDE link (information updated upon request from the client) between Visual Basic (the client) and Q+E (the server).

3. Use LinkRequest to update information in Visual Basic (the client) based on information contained in Q+E (the server).

4. Initiate a hot DDE link (information updated automatically from server to client) between Visual Basic (the client) and Q+E (the server).

5. Use LinkPoke to send information from Visual Basic (the client) to Q+E (the server).

6. Change the LinkMode property between hot and cold.

More Information:

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

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

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

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 client application in Visual Basic.

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

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

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

Default Name	Caption	CtlName
Text1	(N/A)	Text1
Option1	Cold Link	ColdLink

Option2	Hot Link	HotLink
Command1	Poke	Poke
Command2	Request	Request

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

```
Const TRUE = -1
Const FALSE = 0
Const HOT = 1
Const COLD = 2
Const NONE = 0
```

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)
      'Process Windows events. This ensures that Q+E will be executed before any attempt is
              made to perform DDE with it.
    z% = DoEvents ()
      'Clears DDE link if it already exists
    Text1.LinkMode = NONE
      'Sets up link with Q+E.
    Text1.LinkTopic = "QE|QUERY1"
      'Set link to first cell on spreadsheet
    Text1.LinkItem = "R1C1" .
      'Establish a cold DDE link.
    Text1.LinkMode = COLD
    ColdLink.Value = TRUE
End Sub
```

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

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

6. Add the following code to the Clink event procedure of the Hot Link button:

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

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

```
Sub Request_Click ()
        'With a cold DDE link this button will be visible and when selected it will request an update of
                information from the server application to the client 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 client application to the server application.
    Text1.LinkPoke
End Sub
```

You can now run the Visual Basic client application from the Visual Basic 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.

4. Run the application (from Windows if an .EXE file or from the Run menu if from the Visual Basic environment). Form1 of the client 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 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 text box.

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

3. Type some text in the text box in the Visual Basic 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 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 manual.

DDE Example Between Visual Basic and Windows Program Manager

Article Number: Q76551

Summary:

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

The interface commands available through DDE with 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 Program Manager by issuing a LinkRequest to the "PROGMAN" item.

More Information:

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

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

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

Default Name	CtrlName	Caption
TextBox	Text1	
Button	Command1	Make
Button	Command2	Delete
Button	Command3	Request

3. Add the following code to the Click event of Command1:

```
Sub Command1_Click ()
    Text1.LinkTopic = "ProgMan|Progman"
      'Establish Cold link.
    Text1.LinkMode = 2
    Text1.LinkExecute "[CreateGroup(Test Group)]"
      'Make a group in 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.
      'Disconnecting link with Program
    On Error Resume Next
      'Manager causes an error
    Text1.LinkMode = 0r.
      'This is a known problem with Program Manager.
End Sub
```

4. Add the following code to the Click event of Command2:

```
Sub Command2_Click ()
    Text1.LinkTopic = "ProgMan|Progman"
      'Establish Cold link.
    Text1.LinkMode = 2
    Text1.LinkExecute "[DeleteGroup(Test Group)]"
      'Delete the group and all items within it.
      'Disconnecting link with Program
    On Error Resume Next
      'Manager causes an error.
    Text1.LinkMode = 0
      'This is a known problem with Program Manager.
End Sub
```

5. Add the following code to the Click event of Command3:

```
Sub Command3_Click ()
    Text1.LinkTopic = "ProgMan|Progman"
    Text1.LinkItem = "PROGMAN"
      'Establish Cold link.
    Text1.LinkMode = 2
      'Get a list of the groups
    Text1.LinkRequest
      'Disconnecting link with Program
    On Error Resume Next
      'Manager causes an error.
    Text1.LinkMode = 0
      'This is a known problem with Program Manager.
End Sub
```

5. 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 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 3.0 Software Development Kit (SDK) manual mentioned above, the **ExitProgram**() command will work only if Windows Program Manager is NOT the shell (the startup program when you start Windows).

Visual Basic and DDE/OLE with Other Windows Applications

Article Number: Q76562

Summary:

Microsoft Visual Basic 1.0 for Windows can link to a number of Windows applications through dynamic data exchange (DDE). It can also, through the addition of custom controls, link to other Windows applications through object linking and embedding (OLE).

More Information:

Visual Basic has built-in support for DDE. Visual Basic can link and share information with any other Windows application that also supports DDE.

Additional articles in this Help file discuss exactly how to establish a DDE link between Visual Basic and the following applications:

- Another Visual Basic application
- Microsoft Word for Windows
- Microsoft Excel for Windows
- Q+E (shipped with Microsoft Excel)

A Visual Basic application can also use OLE to link with any other Windows application that supports OLE.

Note: This feature is not built into Visual Basic itself.

With the Visual Basic Control Development Kit (CDK)and either the Microsoft Windows Software Development Kit (SDK) and Microsoft C or Microsoft QuickC for Windows, you can create a custom control that supports OLE and you can add this custom control to your Visual Basic application.

Below is a list of applications for Microsoft Windows and their ability to support DDE and/or OLE:

Product	DDE	OLE
Bookshelf 1.0	No	Yes
Money 1.0	No	Yes
Publisher 1.0	No	Yes
Visual Basic 1.0	Yes	No*
Excel 3.0	Yes	Yes
PowerPoint 2.0	No	Yes
Project 1.0	No	No
Word 1.0	Yes	No
Word 2.0	Yes	Yes
Works 2.0	No	No

* Not built in, but can be provided through a Visual Basic custom control.

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

Article Number: Q78165
Modified: 19-DEC-1991

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 were 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
    Const NULL = 0&
    Printer.Print ""
    x% = Escape(Printer.hDC, SETCOPYCOUNT, Len(I%), 0, 3)
    Printer.Print " Printing three copies of this"
```

```
    Printer.EndDoc
End Sub
```

Two Forms, VB.EXE Properties Bar May State Incorrect Focus

Article Number: Q78210
Modified: 15-NOV-1991

Summary:

The FormName property and focus sometimes are not updated correctly when the focus is transferred between forms. This problem occurs at design time in the VB.EXE environment if two or more forms are present and at least one form does not contain any controls. To duplicate the problem, first set the focus to a control on one form and then set the focus to the empty form. While the focus is on the empty form, select the FormName property from the Properties bar. If you now shift the focus from the form with no controls to a control on the other form, the Properties bar will incorrectly state that the focus remains on the original form (the form with no controls).

The problem occurs only when you select a control on one form and shift the focus to the blank form by clicking the mouse.

More Information:

Steps to Reproduce Problem

1. From the VB.EXE File menu, choose New Project.

2. From the File menu, choose New Form.

3. Resize the two forms (Form1 and Form2) so both forms can be viewed at the same time.

4. Place a command button on Form2.

   Note: The focus should remain on the command button.

5. Click Form1 (the form with no controls).

6. From the Properties bar, select FormName. The FormName's Settings box will read "Form1".

7. Click the command button on Form2. At this point, the Properties bar is incorrect, because it is still set to FormName and the Settings box incorrectly reads "Form1".

Workaround

To make the Settings box read "Form2", click Form2 itself instead of clicking the command button on Form2. You can then click the command button to obtain the correct value in the Settings box.

VB CDK VBAPI.LIB Contains CodeView Information; Alternative

Article Number: Q78211
Modified: 15-NOV-1991

Summary:

The Microsoft Visual Basic Control Development Kit (CDK) provides a library of Visual Basic API functions, VBAPI.LIB, which contains Microsoft CodeView information that may not be usable by non-Microsoft languages.

Lstrcpy API Call to Receive LPSTR Returned from Other APIs

Article Number: Q78304
Modified: 20-DEC-1991

Summary:

Because Microsoft Visual Basic does not support a pointer data type, you cannot directly receive a pointer (such as a LPSTR) as the return value from a Windows API or dynamic-link library (DLL) function. You can work around this limitation by receiving the return value as a long integer data type and then using the **lstrcpy**() Windows API function to copy the returned string into a Visual Basic string.

More Information:

An LPSTR Windows API data type is actually a far pointer to a null-terminated string of characters. Because an LPSTR is a far pointer, it can be received as a 4-byte data type, such as a Visual Basic long integer. Using the Visual Basic "ByVal" keyword, you can pass the address stored in a Visual Basic long integer back to the Windows API routine **lstrcpy**() to copy the characters at that address into a Visual Basic string variable. Because **lstrcpy**() expects the target string to be long enough to hold the source string, you should pad any Visual Basic string that you will be passing to **lstrcpy**() so that it will be large enough to hold the source string. Failure to allocate enough space in the Visual Basic string may result in an "Unrecoverable Application Error" (UAE) message when you call **lstrcpy**().

The following program example demonstrates using **lstrcpy**() to retrieve a LPSTR returned from the Windows API routine **GetDOSEnvironment**(). Note that the capability of the Windows API routine **GetDOSEnvironment**() is already available through the Environ function that is built into Visual Basic; therefore, the following program is useful only as an example of using **lstrcpy**().

```
' Global Module declarations
Declare Function GetDOSEnvironment Lib "Kernel" () As Long
Declare Function lstrcpy Lib "Kernel" (
        ByVal lpString1 As Any,
        ByVal lpString2 As Any) As Long

'Form Click event code
Sub Form_Click()
    Dim lpStrAddress As Long,   DOSEnv$
      'Allocate space to copy LPSTR into
    DOSEnv$ = Space$(4096)
      'Get address of returned LPSTR into a long integer
    lpStrAddress = GetDOSEnvironment()
      'Copy LPSTR into a Visual Basic string
    lpStrAddress = lstrcpy(DOSEnv$, lpStrAddress)
      'Parse first entry in environment string and print
    DOSEnv$ = RTrim$(Left$(DOSEnv$, Len(DOSEnv$) - 1))
    Form1.Print DOSEnv$
End Sub
```

Disabling the ENTER Key Beep in a Visual Basic Text Box

Article Number: Q78305
Modified: 15-NOV-1991

Summary:

In a Visual Basic 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 then 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 occurs because a nonmultiline text box is a Windows default edit box class.

Example

The following code will prevent the beep:

```
        ' (Multiline property set to False)
Sub Text1_KeyPress (KeyAscii as Integer)
    If KeyAscii=13 Then
        KeyAscii=0
    End If
End Sub
```

Scope of Line Labels/Numbers in Visual Basic for Windows

Article Number: Q78335
Modified: 15-NOV-1991

Summary:

Line labels (and line numbers) do not follow the same scoping rules as variables and constants in Visual Basic. Line labels must be unique within each module and form. However, you can transfer control only 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" manual or in the Visual Basic online Help system.

Nonstandard Icons Can Make Visual Basic Hang or Display UAE

Article Number: Q78380
Modified: 5-DEC-1991

Summary:

If you create an icon in other than the standard .ICO format and attach that icon to a Visual Basic form as the icon that is displayed during minimization or pasted directly to the form, you may get an "Unrecoverable Application Error" (UAE) message or the machine may hang. Nonstandard icons can also cause less severe run-time error messages such as "Invalid Picture." The icon will load at design time but cause problems at run time.

More Information:

Icons created with utilities other than IconWorks (even the Windows Software Development Kit [SDK] Paint utility) can cause problems because they may not conform to the standard .ICO format. The standard .ICO format that Visual Basic supports is a 32-by-32 pixel matrix, which is specified in the icoDIBSize field in the header of the resource file. Because icons are handled as resources, they can cause problems once they are incorporated into the .EXE file. For example, icons can actually corrupt the code, leading to a hang during execution or causing Windows to generate a UAE message.

References

How to Subclass a VB Form Using VB CDK Custom Control

Article Number: Q78398
Modified: 13-DEC-1991

Summary:

In Windows programming terms, subclassing is the process of creating a message handling procedure and intercepting messages for a given window, handling any messages you choose, and passing the rest to the window's original message handler.

The subclass procedure is a message filter that performs nondefault processing for a few key messages and passes other messages to a default window procedure using **CallWindowProc**(). The **CallWindowProc**() function passes a message to the Windows system, which in turns sends the message to the target window procedure. The target window procedure cannot be called directly by the subclass procedure because the target procedure (in this case a window procedure) is exported.

Below is a simple example of how to subclass a Visual Basic form by writing a custom control with the Visual Basic Control Development Kit (CDK).

More Information:

The following code example demonstrates how to subclass a form from a custom control using the CDK.

This example is developed using the CIRCLE.C program example 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 as well as a C compiler capable of creating a Windows 3.0 compatible dynamic-link library (DLL).

The basic idea for subclassing is to directly examine the window structure of a window, using **GetWindowLong**() to determine the address of the original window procedure. You can then change the address of the target window's window procedure to the address of your subclass procedure using **SetWindowLong**(). In your subclass window procedure, you handle the messages you want and use **CallWindowProc**() to pass the other messages to the original window procedure.

```
        //=================== CIRCLE1 ==================
        // CIRCLE.C
        // An example of subclassing a Visual Basic 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 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

VB CDK Custom Property Name Cannot Start with Numeric

Article Number: Q78399
Modified: 13-DEC-1991

Summary:

The Property Name (npszName) field in the PROPINFO structure for the Visual Basic Control Development Kit (CDK) cannot start with a numeric value.

This information needs to be added to page 132 of the "Microsoft Visual Basic: Control Development Guide" add-on for Microsoft Visual Basic programming system version 1.0 for Windows.

More Information:

When a controls property starts with a numeric value, Visual Basic will generate the binding/syntax checking error "Expected: end-of-statement." However, the property works correctly in the Visual Basic (design) mode from the Properties bar.

To reproduce the error message, do the following:

1. Rebuild the Circle3 example provided with the CDK after changing the PROPINFO Property_FlashColor structure in CIRCLE3.H to the following:

   ```
   PROPINFO Property_FlashColor = { "2FlashColor", DT_COLOR | PF_fGetData | PF_fSetData |
           PF_fSaveData | PF_fEditable, OFFSETIN(CIRCLE, FlashColor) } ;
   ```

2. While in Visual Basic development environment (VB.EXE) with the Circle3 control loaded, assign the 2FlashColor a value.

   ```
   Circle1.2FlashColor = 2
   ```

3. Press F5 to generate the "Expected: end-of-statement" error message. The text "FlashColor" will be selected for the syntax error.

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

Article Number: Q78478
Modified: 20-DEC-1991

Summary:

Command buttons in Visual Basic are limited to a single line of text and one background color (gray). Visual Basic offers no provision for changing colors or displaying bitmaps within a command button to alter its appearance. However, by using a picture control and manipulating the DrawMode in conjunction with the Line method, you can create the look and feel of a command button. 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 farther 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.

To create a customized "command button":

1. Start Visual Basic, or choose New Project from the File menu (ALT, F, N) if Visual Basic 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. 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 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.

VB "UAE" Adding Form with Control Having Same Name as Form

Article Number: Q78564
Modified: 20-DEC-1991

Summary:

If you add a form to a project and that form contains a control with the same name as an existing form, Windows 3.0 will display an "Unrecoverable Application Error" (UAE) message when you try to view the form you just added.

More Information:

Steps to Reproduce Problem

1. Add a text box to the form (Text1 will be the name of the text box by default).

2. From the File menu, choose Save to save the form as Form1.

3. From the File menu, choose New Project.

4. On the Properties bar, select FormName to change the name of the form to Text1.

5. From the File menu, choose Add File and select Form1 (the file you saved in step 1).

6. Choose the OK button in response to the "'Text1' is a form name" message.

7. Select the newly added form (Form1) in the project window.

8. Choose the View Form button.

An "Unrecoverable Application Error" message will be displayed.

CTRL+LEFT/RIGHT ARROW Different in Editor vs. Immediate Window

Article Number: Q77928
Modified: 2-DEC-1991

Summary:

The key combinations CTRL+LEFT ARROW and CTRL+RIGHT ARROW work differently when editing code in a procedure than when typing in the Immediate window.

In the Immediate window, CTRL+LEFT ARROW will move the cursor in front of the preceding word even if that word is one of the following symbols:

! @ # $ % ^ ^ & * ( ) { } : ; , " ' [ ] < >

In the code editor, these symbols are not treated as words; therefore, the cursor will skip over them when the arrow key combinations are used to position the insertion point.

More Information:

In a code window, using the LEFT ARROW key with the CTRL key held down will move the cursor to the beginning of the preceding word or letter on that line, disregarding any punctuation marks and other symbols (that is, any character obtained by typing a number while holding down the SHIFT key, all punctuation marks, brackets, braces, and single and double quotation marks.

In the Immediate window, only the period is not treated as a word and is skipped over when using the CTRL+LEFT ARROW or CTRL+RIGHT ARROW key combination.

Steps to Reproduce Problem

1. Run Visual Basic.

2. From the File menu, choose New Project (ALT, N, P). Form1 will be created by default.

3. Press F7 or double-click Form1 to bring up the Code window.

4. Enter the following code in the Form_Click event procedure of Form1:

```
Sub Form_Click()
    Print "Home."
End Sub
```

5. While the cursor is still at the end of the line, press CTRL+LEFT ARROW to move the cursor to the beginning of the previous word. The cursor should move directly in front of the "H" in Home.

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

7. Press CTRL+BREAK to bring up the Immediate window.

8. Type the following in the Immediate window:

```
Print "Home."
```

9. With the cursor at the end of the line, press CTRL+LEFT ARROW. The insertion point should be directly in front of the last double quotation mark.

In VB, Clipboard.SetData Gives "Invalid Clipboard Format" with Icon

Article Number: Q78073
Modified: 19-NOV-1991

Summary:

If you use the Visual Basic LoadPicture function to load an icon file (.ICO) into a picture control, then attempt to copy that picture control's picture to the Clipboard using the SetData method, the error message "Invalid Clipboard Format" will be displayed, regardless of the format specified for the SetData method.

More Information:

This error will also occur if you attempt to directly load an icon file into the Clipboard using:

```
ClipBoard.SetData LoadPicture("c:\vb\icons\arrows\arw01rt.ico")
```

To work around this problem, set the picture control's AutoRedraw property to True (-1) and use the Picture control's Image property in the SetData method rather than the Picture control's Picture property.

```
    'This code will fail with the error "Invalid Clipboard Format"
Picture1.Picture = LoadPicture("c:\vb\icons\arrows\arw01rt.ico")
Clipboard.SetData Picture1.Picture, 2

    'This will avoid the error
Picture1.Autoredraw = -1
Picture1.Picture = LoadPicture("c:\vb\icons\arrows\arw01rt.ico")
Clipboard.SetData Picture1.Image, 2

    'This will also work
Picture1.Picture = LoadPicture("c:\vb\icons\arrows\arw01rt.ico")
Picture1.Picture = Picture1.Image
Clipboard.SetData Picture1.Picture, 2
```

Restart in VB Break Mode If Delete Blank Line Above End Sub

Article Number: Q78074
Modified: 13-DEC-1991

Summary:

Deleting a blank line above the End Sub/End Function or below the Sub/Function statement will generate the message

You will have to restart your program after this edit--proceed anyway?

while in break mode in the VB.EXE environment. This behavior is by design.

More Information:

Deleting the line following the Sub or Function statement requires you to restart when in break mode. This also occurs when deleting the line preceding the End Sub or End Function statement of any procedure. The Visual Basic edit manager treats both of these deletions as modifications to the first or last lines, both of which require a restart when in break mode.

To force a restart in a program while in break mode:

1. In a new project, choose Start from the Run menu.

2. Press CTRL+BREAK to suspend execution of the application and enter break mode.

3. Press F7, or from the Code menu, choose View Code to bring up the code window.

4. The text cursor should be on the blank line between the following procedure statements:

```
Sub Form_Click ()

End Sub
```

5. Press DEL to delete the blank line between the Sub Form_Click() and End Sub lines.

The following message will be displayed:

You will have to restart your program after this edit--proceed anyway?

The above message is also displayed when the cursor is on the second line and you press the BACKSPACE key once, or if the cursor is at the beginning of the last line of a procedure (at the beginning of the End Sub line) and you press the BACKSPACE key once.

"Printer Error" Printing VB Form to Text-Only Printer

Article Number: Q78075
Modified: 13-DEC-1991

Summary:

The message "Printer Error" is displayed when you print a form from Visual Basic to a text-only printer. The text-only printer does not have the graphics capability to print the Visual Basic form, and Windows 3.0 traps the printer error and displays the "Printer Error" dialog box. This behavior is by design.

More Information:

Steps to Reproduce Problem

1. From the Windows Control Panel, choose the Printers icon, and select Generic Text / Text Only as the default printer. (You may need to install the Generic / Text Only printer from the Control Panel to make this option available.)

2. Start Visual Basic.

3. From the File menu, choose Print. The current form and code are selected by default in the print dialog box.

4. Choose the OK button to print. Windows 3.0 displays the "Printer Error" dialog box.

Printing VB Source w/ HPPCL5A.DRV to HP LaserJet III Cuts Line

Article Number: Q78079
Modified: 15-NOV-1991

Summary:

Using the Visual Basic File menu Print command to print source code will truncate one line of code per page of output when printing to a Hewlett-Packard (HP) LaserJet Series III printer using the HPPCL5A.DRV printer driver. This is a problem with the HP LaserJet Series III printer driver version 3.42 for Windows.

Microsoft has confirmed this to be a problem with the HPPCL5A printer driver version 3.42.

This problem was corrected by the HP III driver version 30.3.85 included with Microsoft Word for Windows version 2.0.

Case Change in VB.EXE Control Name Ignored in Code

Article Number: Q78131
Modified: 15-NOV-1991

Summary:

The capitalization of a control's name (CtlName) can be changed from the Properties bar. However, the change does not take effect within the code unless the name is first changed to something different, then back to the desired CtlName (for example: "Command1" changed to "Test," then changed to "COMMAND1").

More Information:

Steps to Reproduce Problem

1. Start Visual Basic.

2. Place a control button on the default form (Form1).

3. From the Properties list box on the Properties bar, select CtlName.

4. Double-click the control button to see the code for that button.

5. Change the "Command1" default control name to "COMMAND1" on the Properties bar.

6. Double-click the control button again. Note that the capitalization of Sub Command_Click () is the same. Sub Command1_Click () remains Sub Command1_Click () while the CtlName is "COMMAND1" (without the quotation marks).

7. Change the CtlName property from "COMMAND1" to "Test."

8. Double-click the control button. Note that SUB Command1_Click () changed to SUB Test_Click () as it should.

9. Change the CtlName property from "Test" to "COMMAND1." Sub Test_Click () should now be Sub Command1_Click (), the desired CtlName with the proper capitalization.

Capitalization changes are not updated in the code when they are changed on the Properties bar. Only changes in the number of letters in the CtlName force an update of the code associated with the form or control.

ToolBox Picture Control Bitmap Too Small on EGA

Article Number: Q78132
Modified: 15-NOV-1991

Summary:

The bitmap for the picture control (in the ToolBox window) in EGA mode is 27-by-22 pixels, when it should be 28-by-22 pixels. The result is a black line 2 pixels thick at the left side of the picture control bitmap, rather than 1 pixel thick as it should be.

Problems

DateSerial Does Not Give Error for Invalid Month or Day

Date/Time Functions Give Wrong Results After CURRENCY Format$

Disabling Menu Item with Submenus Can Disable Shortcut Keys

VB Dynamic Drive Control Arrays Don't Get Correct Coordinates

Compatibility Problems with Adobe Type Manager (ATM) and VB

DEL Key Behavior Depends on Text Box MultiLine Property

"Not Enough Memory to Load Tutorial" with Corrupt VB.LES File

CTRL+LEFT/RIGHT ARROW Different in Editor vs. Immediate Window

"Printer Error" Printing VB Form to Text-Only Printer

Printing VB Source w/ HPPCL5A.DRV to HP LaserJet III Cuts Line

Nonstandard Icons Can Make Visual Basic Hang or Display UAE

VB "UAE" Adding Form with Control Having Same Name as Form

Right Mouse Button Causes Remote Control Menus in Visual Basic

In VB, Format$ Using # for Digit Affects Right Justification

VB Multiline Text Box Memory Not Freed When Form Is Unloaded

VB Extra Resize Event When WindowState Not Normal (Not 0)

Text Too Narrow with Italic Fonts in Visual Basic Labels

DateSerial Does Not Give Error for Invalid Month or Day

Article Number: Q77393
MODIFIED: 28-OCT-1991

Summary:

The DateSerial function doesn't generate an error when you use values for the month and the day arguments outside the ranges specified in the "Microsoft Visual Basic: Language Reference" version 1.0 manual. In addition, you can use a numeric expression for each argument representing the number of days, months, or years before or after a certain date.

However, you get an "Illegal function call" error message if you use a value for the year that is not between 1753 and 2078 (inclusive). You also get the error if the date specified by the three arguments either directly or indirectly evaluates to a date that is before January 1, 1753 or after December 31, 2078.

More Information:

Page 65 of the "Microsoft Visual Basic: Language Reference" version 1.0 manual states the following:

> ...the range of numbers for each DateSerial argument should conform to the accepted range of values for the unit. These values are 1 to 31 for days, and 1 through 12 for months. You can also specify relative dates for each argument by using numeric expressions representing the number of days, months, or years before or after a certain date...

You can actually have values outside these ranges for the month and day argument and Visual Basic will not give an error. For example, a value of 0 for the day evaluates to the last day of the previous month. A value of 13 for the month translates to the first month (January) of the next year.

The following are examples of statements that will not produce errors:

```
x# = DateSerial(63,7,12)      'evaluates to July 12, 1963
x# = DateSerial(63,13,5)      'evaluates to January 5, 1964
x# = DateSerial(63,7,33)      'evaluates to August 2, 1963
x# = DateSerial(63,10,-1)     'evaluates to September 29, 1963
x# = DateSerial(63,-1,5)      'evaluates to November 5, 1962
```

The following statements will generate an "Illegal function call" error because they produce dates before January 1, 1753, and after December 31, 2078:

```
x# = DateSerial(1750,3,1)     'evaluates to March 1, 1750
x# = DateSerial(2078,12,40) 'evaluates to January 9, 2079
x# = DateSerial(1753,-5,20) 'evaluates to July 20, 1752
```

Date/Time Functions Give Wrong Results After CURRENCY Format$

Article Number: Q77579
MODIFIED: 30-OCT-1991

Summary:

The Year, Month, Day, Weekday, Hour, Minute, and Second functions give incorrect results just after using the Format$ function on a CURRENCY data type. The serial number used is correct, but the aforementioned functions using that serial number produce incorrect results.

Microsoft has confirmed this to be a problem in Microsoft Visual Basic programming system version 1.0 for Windows.

More Information:

To work around this problem, invoke the Format$ function again using a non-CURRENCY data type. The Year, Month, Day, Weekday, Hour, Minute, and Second functions will then work correctly.

To illustrate the problem, attach the following code to a form:

```
Sub Form_Click ()
    Serial# = Now
    Print
    Print Format$(0@, "0.00")
    Print Serial#
    Print Year(Serial#)
    Print Month(Serial#)
    Print Day(Serial#)
    Print Weekday(Serial#)
    Print Hour(Serial#)
    Print Minute(Serial#)
    Print Second(Serial#)
End Sub
```

The Year, Month, Day, Weekday, Hour, Minute, and Second functions will produce incorrect results.

To obtain the correct results, format another data type and assign it to a dummy variable before using the date/time functions. In the sample code above, attach the following statement after "Print Format$(0@, "0.00")" and before "Print Year(Serial#)":

```
dummy$ = Format$(0, "0.00")
```

This statement will cause the Year, Month, Day, Weekday, Hour, Minute, and Second functions to produce the correct results.

Disabling Menu Item with Submenus Can Disable Shortcut Keys

Article Number: Q77581
MODIFIED: 30-OCT-1991

Summary:

If a menu item containing one or more submenus is disabled during run time, the shortcut keys of subsequent menu items on the same level of the same parent menu will no longer function. If the menu item is disabled during design time (in the Menu Design window), all shortcut keys will function properly at run time.

More Information:

Steps to Reproduce Problem

1. Run Visual Basic. From the File menu, select New Project (ALT, F, N). Form1 will be created by default.

2. From the Menu Design window, create the following menu structure on Form1:

```
Menu1
    Item1
        Item1Subitem1
    Item2
```

Add the shortcut CTRL+I (or any other shortcut key) to the menu item Item2. (To assign a function-key or control-key shortcut to a menu command, select from the list in the Accelerator drop-down list box in the Menu Design window. See page 120 of the "Visual Basic: Programmer's Guide" version 1.0 manual for more information.)

The actual menu on Form1 should look like this:

```
Menu1
---------
Item1        >
Item2     CTRL + I
```

3. Place a command button on Form1.

4. Place the following code in the Command1_Click event procedure:

```
Sub Command1_Click ()
    Item1.Enabled = 0
End Sub
```

5. Place the following code in the Item2_Click event procedure:

```
Sub Item2_Click ()
    Beep
    Beep
    Beep
End Sub
```

6. Press F5 to run the program.

7. From Menu1, choose Item2 by pressing the CTRL+I shortcut key combination. The program should respond by beeping the computer's internal speaker three times.

8. Click the command button labeled Command1.

   Note: Item1 in Menu1 will now be disabled.

9. From Menu1, choose Item2 by pressing the CTRL+I key combination. The program will NOT respond correctly (with three beeps). The menu shortcut no longer causes an Item2_Click event to occur.

   Note: If Item2 is selected from Menu1 with the mouse, the program will respond correctly.

VB Dynamic Drive Control Arrays Don't Get Correct Coordinates

Article Number: Q77644
MODIFIED: 9-DEC-1991

Summary:

Setting the Left, Width, or Top coordinate of the first element of a control array of dynamic drive controls then loading a second element of the array will result in the Left, Width, or Top coordinate of the first and second elements being different. This problem occurs when ScaleMode property of the form is set to Twip, Point, Millimeter, Centimeter, or User ScaleMode. The problem does not occur when the ScaleMode property of the form is set to Character, Pixel, or Inch.

More Information:

When the ScaleMode property of the form is set to Character, Pixel, or Inch, the second and subsequent elements of the control array maintain the same coordinates as the first element of the array, as expected. When the ScaleMode of the form is set to Twip, Point, Centimeter, Millimeter, or User, the subsequent elements of the control array of dynamic drive controls will not have the same coordinates as the first element of the array.

Steps to Reproduce Problem

1. Start Visual Basic. From the File menu, choose New Project (ALT, F, N).

2. Place a drive control (Drive1) on a blank form (Form1).

3. From the Properties bar, set the Index property of Drive1 to 0 (this is the first element of a control array).

4. Double-click anywhere on the form to bring up the Form_Click event procedure.

5. Add the following code to the Form_Click event procedure of Form1:

```
SUB Form_Click ()
        'Set the x coordinate of the first element of the array
    Drive1(0).Left = 2000
      'This loads the second element of the control array
    Load Drive1(1)
      'Print the x coordinate of the two elements of   the control array
    Print Drive1(0).Left,Drive1(1).Left
End Sub
```

6. From the Run menu, choose Start.

7. Click anywhere on the form.

The x (left) coordinates of the Drive1(0) and drive Drive1(1) will print directly on the form.

Note: The Drive1(0) left coordinate will be 2000, where the Drive1(1) left coordinate is 1995.

A similar discrepancy occurs when the ScaleMode property of Form1 is set to Point, Centimeter, Millimeter, or User.

Specifying Drive1(0).Width then loading the next control array element results in the same problem. The width of the second element of the array will be slightly different from the first.

The Align to Grid feature of the Visual Basic 1.0 programming environment (VB.EXE) does not change the position or size of any of the elements of the control array.

This problem does not occur when the ScaleMode property of the form is set to Character, Pixel, or Inch.

A workaround for the problem is to set the Left, Top, or Width property of the second control array element equal to the like property of the first control array element. Insert the following code after the Load Drive1(1) statement in step 5 above to work around the problem as follows:

```
Drive1(1).Left = Drive1(0).Left
```

Compatibility Problems with Adobe Type Manager (ATM) and VB

Article Number: Q77645
MODIFIED: 31-OCT-1991

Summary:

The following problems may arise when using Adobe Type Manager (ATM) with Microsoft Visual Basic programming system version 1.0 for Windows:

- FontName list is incorrect and/or contains duplicate names.

- "Unrecoverable Application Error" (UAE) messages.

- Incorrect screen font displayed when using ATM fonts.

The product included here, Adobe Type Manager, is manufactured by vendors independent of Microsoft; we make no warranty, implied or otherwise, regarding this product's performance or reliability.

DEL Key Behavior Depends on Text Box MultiLine Property

Article Number: Q77737
MODIFIED: 16-DEC-1991

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 3.0 and is not specific to Microsoft Visual Basic.

More Information:

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
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.

"Not Enough Memory to Load Tutorial" with Corrupt VB.LES File

Article Number: Q78000
MODIFIED: 19-NOV-1991

Summary:

If you try to run the Visual Basic tutorial when it is not actually installed or the file VB.LES has become corrupted, a message box will state "Not Enough Memory To Load Tutorial." A workaround is described farther below.

More Information:

The subdirectory VB\VB.CBT contains files for the Visual Basic tutorial. If the file VB.LES has been modified or replaced by another file, the tutorial cannot be run and two erroneous dialog boxes will open.

The first dialog box has the title "Visual Basic Tutorial" and displays the message "Out of memory." Choosing the OK button will clear this box and another one will open.

The second dialog box is titled "Microsoft Visual Basic," which displays the message "Not enough memory to load tutorial." Choose the OK button to clear this box.

The messages displayed in these dialog boxes are incorrect and should be ignored. To correct this problem, reinstall Visual Basic so that VB.LES will be replaced by the correct file.

> Note: To properly reinstall Visual Basic, you must first delete all files from the previous installation. Remember to save all your program files (*.FRM, *.MAK, and so on) before deleting the previous installation.

Visual Basic Online Knowledge Base

Welcome to the Visual Basic online Knowledge Base. This help system comprises over 200 articles that relate to the Visual Basic programming environment for Windows. These articles were developed by Microsoft's Basic Language Product Support personnel in direct response to questions asked by users and programmers of Visual Basic. New and updated articles can be found on both Compuserve and GENIE. This help file describes helpful tips, critical information, references, and other information that may not be addressed in the manuals.

Please feel free to distribute this information in compliance with the terms of Microsoft.

Advanced Programming Tips	Advanced tips about programming using Visual Basic code and Windows 3.0 API function calls along with advice for converting older Basic code to Visual Basic code.
Dynamic Data Exchange	Dynamic data exchange (DDE) programming examples to Excel, Q+E, Word for Windows and Visual Basic.
Documentation Issues	Issues related to the "Microsoft Visual Basic Language Reference," the "Microsoft Visual Basic Programmer's Guide," and the online Help system.
Error Messages	Puzzling error messages that you may receive when programming in the Visual Basic environment.
CDK Issues	Tips, advice, and corrections for the Custom Control Development Kit (CDK)
Design Issues and Limits	Design issues and specifications of the Visual Basic language.
Problems	Reported problems with Visual Basic.
Service Issues	Service-related issues.
Setup Issues	Issues and problems that you could encounter while setting up Visual Basic.

Setup Issues

The Setup program is designed to install Visual Basic on your computer. The articles below describe situations that may arise when you are installing Visual Basic.

Removing Disk During VB Setup Terminates SETUP, Missing Files

Incomplete Path Causes "Insufficient Disk Space" During Setup

"Insufficient Disk Space" After Setup Begins to Copy Files

"Visual Basic 1.0 Setup by Batch File" Appnote Available

Setup "Illegal Function Call" If Manually Delete Existing File

INFORMATION PROVIDED IN THIS SOFTWARE(collectively referred to as online Knowledge Base) IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE. The user assumes the entire risk as to the accuracy and the use of this online Knowledge Base. This online Knowledge Base may be copied and distributed subject to the following conditions:

1. All files on the disk(s) must be copied without modification (the MS-DOS utility **diskcopy** is appropriate for this purpose);

2. All components of this online Knowledge Base must be distributed together;

3. This online Knowledge Base may not be distributed for profit.

Incomplete Path Causes "Insufficient Disk Space" During Setup

Article Number: Q74646

Summary:

You may receive an "Insufficient disk space on: XXX" error message when running the Microsoft Visual Basic version 1.0 Setup program if you have not specified a drive letter in your path to Windows 3.0.

More Information:

If you have a partial PATH environment variable to the Windows environment (for example, PATH=\WIN3) in your AUTOEXEC.BAT file, you may need to reset the PATH environment variable to C:\WIN3. After you reset the path, Setup should run correctly.

"Insufficient Disk Space" After Setup Begins to Copy Files

Article Number: Q74648

Summary:

If you receive an "Insufficient disk space" error message when running Visual Basic's Setup program, it may be caused by using Windows with a temporary Windows swap file instead of a permanent Windows swap file.

More Information:

Pages 520 through 529 in the "Microsoft Windows User's Guide" version 3.0 manual discuss Windows swap files. Permanent swap files are contiguous so that your disk does not contain files in fragmented pieces, which may occur if you are using temporary swap files. Temporary Windows swap files may grow in size, which may cause the "Insufficient disk space" error during the execution of Visual Basic's Setup program. However, permanent Windows swap files will not change in size, so using permanent Windows swap files may help to avoid the "Insufficient disk space" error.

"Visual Basic 1.0 Setup by Batch File" Appnote Available

Article Number: Q75370

Summary:

In rare cases, certain hardware configurations may not work properly with the SETUP.EXE utility shipped with Microsoft Visual Basic programming system version 1.0 for Windows.

As an alternative installation method for Visual Basic 1.0, you can use the application note titled "Visual Basic 1.0 Setup by Batch File" (BV0446). To obtain application notes, call Microsoft Product Support Services at (206) 454-2030. Please specify whether you want a 3.5-inch (720K) or 5.25-inch (1.2 MB) disk (no other disk format is available).

More Information:

Microsoft can distribute this application note to only registered owners of Visual Basic because it contains uncompressed copies of Visual Basic's Icon Library files. Thus, this application note is not available on online electronic services; it is only available directly from Microsoft.

This application note includes the following files and subdirectories on disk:

README.NOW
VBSETUP.BAT
VBSETUP1.BAT
DECOMP.EXE
\ARROWS
\COMM
\COMPUTER
\FLAGS
\MAIL
\MISC
\OFFICE
\TRAFFIC
\WRITING

VBSETUP.BAT must be run from MS-DOS and from the drive in which you inserted the disk. For example, typing the following from the drive C MS-DOS prompt won't work:

a:vbsetup

Instead, you must make drive A (or the drive in which you inserted the disk) the active drive before running VBSETUP.BAT. For example, type **A:** at the drive C MS-DOS prompt and then type **vbsetup** at the drive A MS-DOS prompt.

> Note: When you run VBSETUP as shown above (with no arguments), further usage instructions will be displayed on the screen. The Icon Library shipped with Visual Basic is contained on the enclosed disk in uncompressed format, and will be copied to your destination directory by VBSETUP. This disk (and the included Icon Library) may be distributed only to registered owners of Microsoft Visual Basic. VBSETUP.BAT will work only with 5.25-inch 1.2 MB disks, or with 3.5-inch 720K disks. VBSETUP.BAT will not work with 5.25-inch 360K disks.

Setup "Illegal Function Call" If Manually Delete Existing File

Article Number: Q75412

Summary:

If the Microsoft Visual Basic Setup program detects an existing copy of Visual Basic already on the disk during installation, Setup will ask whether or not you want to "Overwrite existing files?" At this point, if you manually delete the existing files using a separate process in Windows and then attempt to continue with the Setup, Setup will give an "Illegal function call" error message and end without installing Visual Basic.

To work around the problem, let Setup overwrite the existing files.

More Information:

Steps to Reproduce Problem

1. Have an installation of Visual Basic in the \VB subdirectory of your hard disk.

2. Run the Visual Basic SETUP.EXE program.

3. When the subdirectory path to install Visual Basic is displayed, make sure it points to the VB subdirectory where the files already exist. Click Continue.

4. Do not reply to the following message:

   A previous installation of Visual Basic has been found in this directory. Overwrite existing files?

5. Open the File Manager in Windows and delete the existing Visual Basic files.

6. Return to the Visual Basic Setup and click Yes in reply to the message.

When you reply OK to the "Illegal function call" message that appears, Setup ends without installing Visual Basic. To work around this behavior when Visual Basic files already exist prior to installation, either delete those files before starting Setup or let Setup overwrite the existing files.

IconWorks

IconWorks is an acclaimed icon editor program written with Visual Basic 1.0. Source code is supplied to give the user extensive how-to help with programming in Visual Basic using standard code and API function calls. Source code for IconWorks is supplied with Visual Basic 1.0.

Advanced Programming Tips

Data	Topics related to data structures
Forms	Topics related to the Visual Basic Form
List Box	Topics related to the List Box control
Graphics	Information about items related to graphics, such as API calls, AutoRedraw, and bitmaps
Text Box	Topics related to the Text Box control
Window	Topics related to the Windows environment
Miscellaneous	Various topics, such as converting QuickBasic programs, communications, and menus

Advanced Data Tips

How to Pass One-Byte Parameters from VB to DLL Routines

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

How to Get "HWnd" Handle for a Control by Using **GetFocus** API

WINAPI.TXT: Windows API Declarations and Constants for VB

Correction to WINAPI.TXT for Visual Basic Add-On Kit

Advanced Form Tips

How to Create a Flashing Title Bar on a Visual Basic Form

How to Create a System-Modal Program/Window in Visual Basic

How to Invoke **GetSystemMetrics** Windows API Function from VB

Advanced List Box Tips

How to Clear a VB Combo Box with a Windows API Function

How to Set Tab Stops Within a List Box in Visual Basic

How to Clear a VB List Box with a Windows API Function

Visual Basic List Box Won't Open If Resized at Run Time

Advanced Graphics Tips

How to Flood Fill in VB Using Windows API Function Call

How to Use Windows 3.0 **BitBlt** Function from Visual Basic

How to Implement Bitmaps Within Visual Basic Menus

ExtFloodFill Won't Fill Over QBColors(1-8) if AutoRedraw = True

How to Print a VB Picture Control Using Windows API Functions

How to Create Flashing/Rotating Rubber-Band Box in VB

How to Create Rubber-Band Lines/Boxes in Visual Basic

Advanced Text Box Tips

How to Make a VB Text Box Control with a Password (*) Style

How to Limit User Input in VB Combo/Text Box; **SendMessage** API

Determining Number of Lines in VB Text Box; **SendMessage** API

How to Scroll VB Text Box Programatically and Specify Lines

Using Windows API Functions to Better Manipulate Text Boxes

Multiline Text Box Contents Not Grayed When Control Disabled

Disabling the ENTER Key Beep in a Visual Basic Text Box

Advanced Window Tips

Disk Copy of Code Examples from VB Programmer's Guide Manual

Terminating Windows from a Visual Basic Application

VB Can Determine When a Shelled process Has Terminated

How VB Can Determine If a Specific Windows Program Is Running

How to Access Windows Initialization Files Within Visual Basic

How to Determine Multiple Instances of a VB Application

How to Print the ASCII Character Set in Visual Basic

How to Trap VB Form Lost Focus with **GetActiveWindow** API

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

Miscellaneous Advanced Tips

The following articles cover miscellaneous topics that you may find useful.

Program Example for Communication Port Support in Visual Basic

How to Create and Use a Custom Cursor in Visual Basic; Win SDK

How to Create Pop-Up Menus on a Visual Basic Form

How to Emulate Quick Basic's SOUND Statement in Visual Basic

Simulating ON KEY Key Trapping with KeyDown Event in VB

How to Set Hourglass Mouse Pointer in VB Program During Delays

Determining Whether TAB or Mouse Gave a VB Control the Focus

Sending Keystrokes from Visual Basic to an MS-DOS Application

No New Timer Events During Visual Basic Timer Event Processing

Creating Nested Control Arrays in Visual Basic

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

Lstrcpy API Call to Receive LPSTR Returned from Other APIs

Scope of Line Labels/Numbers in Visual Basic for Windows

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

How to Pass One-Byte Parameters from VB to DLL Routines

Article Number: Q71106

Summary:

Calling some routines in dynamic-link libraries (DLLs) requires BYTE parameters in the argument list. Visual Basic 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 can call external code in the form of dynamic-link libraries (DLLs). Some of these libraries require BYTE parameters in the argument list.

GetTempFileName() is documented on page 4-217 of the "Microsoft Windows 3.0 Software Development Kit, Reference - Volume 1." In Visual Basic, 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 why you can use an integer data type for a BYTE data type in these types of procedure calls.

References

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

Article Number: Q71260

Summary:

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

More Information:

The HBITMAP data type represents a 16-bit index to GDI's 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.

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)
```

References

How to Get "HWnd" Handle for a Control by Using GetFocus API

Article Number: Q71799

Summary:

Microsoft Visual Basic does not provide access to the Windows handle (hWnd) for any control other than the Form. By using the Windows API **GetFocus**() function, you can retrieve the handle to most Visual Basic controls.

More Information:

Visual Basic provides a "hWnd" property for the Form, but none of the provided controls has this property. By using the Windows API **GetFocus**() function and the Visual Basic SetFocus method you can obtain the "hWnd" for most Visual Basic controls. With the exception of the Frame and Label controls, neither of these controls can receive the focus for user input and do not support the Visual Basic SetFocus method.

The Visual Basic SetFocus method allows the changing of the input focus to a specific control, rather than waiting for the user to select the desired control manually. The following code changes the focus to the command button "Command1" on the Form:

```
Command1.SetFocus
```

The **GetFocus**() function requires no parameters. The **GetFocus**() function returns an integer value that represents the handle to the control. Use **GetFocus**() to get a handle to the control that currently has the focus:

```
hWd% = GetFocus()
```

Most Windows API functions require the handle (hWnd) of the window that is to be acted upon.

References

WINAPI.TXT: Windows API Declarations and Constants for VB

Article Number: Q73694

Summary:

The file WINAPI.TXT supplies declarations for Microsoft Visual Basic programmers who want to call Windows API routines.

WINAPI.TXT is provided here in the assumption that you already have a reference for Windows API calls, such as the documentation provided with the Microsoft Windows Software Development Kit (SDK).

If you don't have a reference manual for Windows API calls, you can obtain the Visual Basic add-on kit, "Microsoft Windows Programmer's Reference" and OnLine Resource, which includes WINAPI.TXT on disk, available from Microsoft.

More Information:

WINAPI.TXT can be found on CompuServe in the MSLANG forum (GO MSLANG), as well as in the Microsoft Software Library on CompuServe.

Contents of README.NOW

WINAPI.TXT is an ASCII text file containing the functions and constants in the Microsoft Windows 3.0 API, declared in the format used by Microsoft Visual Basic.

To use WINAPI.TXT, you should have the book "Microsoft Windows Programmer's Reference" for Windows version 3.0, or you should have the reference manuals provided with the Microsoft Windows SDK.

WINAPI.TXT includes the following:

- External procedure declarations for all the Microsoft Windows API functions that can be called from Visual Basic.

- Global constant declarations for all the constants used by the Microsoft Windows API.

- Type declarations for the user-defined types (structures) used by the Microsoft Windows API.

WINAPI.TXT is too large to be loaded directly into a Visual Basic module. Attempting to load it directly into Visual Basic will cause an "Out of Memory" error message.

WINAPI.TXT is also too large for the Notepad editor supplied with Microsoft Windows, but it can be loaded by Microsoft Write. To use WINAPI.TXT, load it into an editor (such as Microsoft Write) that can handle large files. Copy the declarations you want and paste them into the global module in your Visual Basic application.

> Note: Some of the Windows API declarations are very long. Some editors will wrap these onto a second line, and will copy them as multiple lines rather than as single lines. Declarations in Visual Basic cannot span lines, so if you paste these as multiple lines, Visual Basic will report an error. If an error occurs, you can either adjust the margins in the editor before copying the lines or remove the line breaks after pasting them.

The global module is the recommended place for the declarations that you copy from the WINAPI.TXT file; however, you can place the external procedure declarations in the Declarations section of any form or module. You can also place the constant declarations anywhere in any module

or form code if you remove the Global keyword. Type declarations must be placed in the global module.

Once you have pasted the declaration for a Windows API routine (as well as any associated constant and type declarations) into your application, you can call that routine as you would call any Visual Basic procedure.

For more information about declaring and calling external procedures, see Chapter 23, "Extending Visual Basic," in the "Microsoft Visual Basic: Programmer's Guide."

Warning: Visual Basic cannot verify the data you pass to Microsoft Windows API routines. Calling a Microsoft Windows API routine with an invalid argument can result in unpredictable behavior: your application, Visual Basic, or Windows may crash, hang, or receive an "Unrecoverable Application Error." When experimenting with Windows API routines, save your work often.

References

Correction to WINAPI.TXT for Visual Basic Add-On Kit

Article Number: Q74526

Summary:

The following corrections apply to the WINAPI.TXT file provided on disk with the "Microsoft Windows Programmer's Reference" book and OnLine Resource, add-on kit number 1-55615-413-5, for Microsoft Visual Basic.

More Information:

The following corrections apply to the WINAPI.TXT file dated 5/14/91.

1. The following statement is incorrect

```
Declare Function SetCommState Lib "User" ()
```

   and should be changed to read as follows:

```
Declare Function SetCommState Lib "User" (lpDCB as DCB) As Integer
```

2. The Declare statement for **WinExec**() is missing the ByVal attribute in the second parameter. Without this ByVal, invoking **WinExec**() can cause a Windows "Unrecoverable Application Error" ("UAE"). The **WinExec**() Declare statement should be changed to read as follows (on one line):

```
Declare Function WinExec% Lib "Kernel" (
        ByVal lpCmdLine$,
        ByVal nCmdShow%)
```

3. The following statement is incorrect

```
Declare GetDC Lib "GDI" (ByVal hWnd As Integer) As Integer
```

   and should be changed to read as follows:

```
Declare GetDC% Lib "USER" (ByVal hWnd%)
```

In other words, the function **GetDC**() is in the Windows "USER" dynamic-link library (DLL), not in the "GDI" library (DLL).

Reference

How to Create a Flashing Title Bar on a Visual Basic Form

Article Number: Q71280

Summary:

When calling a Windows API function, you can create a flashing window title bar on the current form or any other form for which you know the handle.

More Information:

Visual Basic can 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%)

Parameter	Type/Description
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).
Returned Value	**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:

```
Declare Function FlashWindow% Lib "user" (
        ByVal hWnd%,
        ByVal bInvert%)

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

4. In the Form_Load event procedure, add the following code:

```
Sub Form_Load ()
```

```
    Form2.Show
End Sub
```

5. In the Sub_Time1_Timer () procedure of Form1, add the following code:

```
Sub Timer1_Timer ()
    Succ% = FlashWindow(Form1.hWnd, 1)
End Sub
```

6. In the **GetFocus** event procedure of Form1, create the following code:

```
Sub Form_GotFocus ()
    Timer1.Enabled = 0
End Sub
```

7. In the Click event for Form2, add the following code:

```
Sub Form_Click ()
    Form1.Timer1.Enabled = -1
End Sub
```

8. Run the program. Form1 will be in the foreground with Form2 in the background. Click anywhere on Form2; Form1's title bar will flash until you click on Form1.

Reference

How to Create a System-Modal Program/Window in Visual Basic

Article Number: Q72674

Summary:

From a Visual Basic program, you can disable the ability to switch to other Windows programs by calling the Windows 3.0 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 could even remove the Control 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.

> Note: Care must be taking when using the **SetSysModalWindow**() API from within the Visual Basic programmer's 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:

```
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 the focus until that window is destroyed.

References

How to Invoke GetSystemMetrics Windows API Function from VB

Article Number: Q77061

Summary

The Windows 3.0 API function call **GetSystemMetrics**() can return useful information about the Windows system. **GetSystemMetrics**() can be called directly from Visual Basic 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.

More Information:

The Windows 3.0 function call **GetSystemMetrics**() 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 buttons has been changed. System metrics depend on the system display and may vary from display to display.

The Visual Basic declaration for **GetSystemMetrics**() is:

GetSystemMetrics% (ByVal nIndex%)

Parameter	Type/Description
nIndex%	Specifies the system measurement to be retrieved. All measurements are in pixels.
Returned Value	The value returned from the GetSystemMetrics% function --specifies the system metrics.

Below is a sample call to find if the present version of Windows is a debugging version:

```
Declare Function GetSystemMetricsLib "User" (ByVal Param%) As Integer

ScaleMode = 3   'select pixel
Print "Debugging version :   ;
GetSystemMetrics(SM_DEBUG)
```

The constants and meaning for nIndex% are as follows:

Constant/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

References

How to Clear a VB Combo Box with a Windows API Function

Article Number: Q76513

Summary:

This article explains how to instantly clear the contents of a Visual Basic combo box by sending the combo box a CB_RESETCONTENT message.

More Information:

There is no single command within Visual Basic that will clear combo box entries. However, by using the **SendMessage**() function, you can clear out the contents of a combo box with one command. The arguments to **SendMessage**() with the CB_RESETCONTENT parameter are

SendMessage&(hWnd%, wMsg%, wParam%, lParam&)

Parameter	Type/Description
hWnd%	Identifies the window that is to receive the message.
wMsg%	The message to be sent (CB_RESETCONTENT = &H411).
wParam%	Is not used (NULL).
lParam&	Is not used (NULL).

Specifying wMsg% equal to &H411 sends a CB_RESETCONTENT message to the combo box. This removes all strings from the combo box and frees any memory allocated for those strings.

To get hWnd%, the handle to the target window, you must call the Windows API function **GetFocus**(). This method will return the handle to the control that currently has focus. For a combo box with a style property of 2 (drop-down list), this will return the handle to the combo box that you want to send the message to. For other styles of combo boxes, the focus is set to a child edit control that is part of the combo box, and you must use the **GetParent**() Windows API function to get the handle to the combo box itself.

The following steps demonstrate how to delete entries from a combo box:

1. Create a combo box called Combo1 on Form1.

2. Declare the following Windows API functions at the module level or in the Global section of your project:

```
Declare Function SendMessage% Lib "user"(
        ByVal hWnd%,
        ByVal wMsg%,
        ByVal wParam%,
        ByVal lParam&)
Declare Function GetFocus% Lib "user" ()
Declare Function APISetFocus% Lib "user" Alias "SetFocus" (ByVal hWnd%)
Declare Function GetParent% Lib "user" (ByVal hWnd%)
```

   Note: Each Declare statement must be written on one line.

3. Declare the following constants in the same section:

```
Global Const WM_USER = &H400
```

```
Global Const CB_RESETCONTENT = WM_USER + 11
Global Const DROP_DOWN_LIST = 2
```

4. Place some entries in the combo box in your Form_Load procedure:

```
Sub Form_Load ()
    For i = 1 To 10
      'Put something into combo box.
    Combo1.AddItem Format$(i)
    Next
End Sub
```

5. Create a Sub within the (declarations) section of the Form1 code window with the following code:

```
Sub ClearComboBox (Combo As Control)
    hWndOld% = GetFocus()
    Combo.SetFocus
    If Combo.Style = DROP_DOWN_LIST then
        x = SendMessage(GetFocus(), CB_RESETCONTENT, 0, 0)
    Else
        x = SendMessage(GetParent(GetFocus()), CB_RESETCONTENT, 0, 0)
    End If
    Suc% = APISetFocus(hWndOld%)
End Sub
```

6. Within an event procedure, call ClearComboBox with the name of the Combo box as a parameter:

```
Sub Form_Click ()
    ClearComboBox Combo1
End Sub
```

7. Run the program and click anywhere on Form1. This will clear the combo box.

References

How to Set Tab Stops Within a List Box in Visual Basic

Article Number: Q71067

Summary:

Visual Basic does not have an intrinsic function for creating multicolumn list boxes. To create multicolumn list boxes, you must call several Windows API function calls to set tab stops within the list box. The tab stops create the multicolumn effect.

More Information:

Multicolumn looking list boxes can be created by calling several Windows API function calls: **GetFocus**(), **SendMessage**(), and **SetFocus**().

The function **GetFocus**() requires no parameters. This function will return an integer value that represents the handle to the control. Use **GetFocus**() to get the handle to the control that currently has focus upon entry to the event-handler procedure. After you store the handle to the control that currently has focus, set the focus to the desired list box.

After you set the focus to the list box, you must send a message to the window's message queue that will reset the tab stops of the list box. Using the argument LB_SETTABSTOPS as the second parameter to **SendMessage**() will set the desired tab stops for the multicolumn list box based on other arguments to the function. The **SendMessage**() function requires the following parameters for setting the tab stops:

SendMessage (hWnd%, LB_SETTABSTOPS, wParam%, lParam&)

Parameter	Type/Description
hWnd%	Handle to the control that has the focus.
wParam%	An integer that specifies the number of tab stops in the list box.
lParam&	A long pointer to the first member of an array of integers containing the tab stop position in dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to 1/4 of the current dialog base-width unit. The dialog base units are computed based on the height and the width of the current system font. The **GetDialogBaseUnits**() function returns the current dialog base units in pixels.) The tab stops must be sorted in increasing order; back tabs are not allowed.

After setting the tab stops with the **SendMessage**() function, calling **APISetFocus**() with the saved handle will return the focus back to the control that had focus before the procedure call. **APISetFocus**() is the alias for the Windows API **SetFocus**() function. The Windows API **SetFocus**() function needs to be redefined using the "Alias" keyword because "SetFocus" is a reserved word within Visual Basic.

To create multicolumn list boxes within Visual Basic, create a list box named List1 on Form1. Declare the following Windows API functions at the module level or in the Global section of your code as follows:

```
Declare Function GetFocus Lib "user" () As Integer
Declare Function SendMessage Lib "user" (
        ByVal hWnd As Integer,
        ByVal wMsg As Integer,
        ByVal wp As Integer,
```

```
            lp As Any) As Long
Declare Function APISetFocus Lib "user" Alias "SetFocus" (
            ByVal hWnd%) As Integer

        'Note: All Declare statements must each be written on one line.

Const WM_USER = &H400
Const LB_SETTABSTOPS = WM_USER + 19
```

Include the following code within a Sub procedure:

```
Sub Form_Click ()
    Static tabs(3) As Integer
      'Remember who had the focus.
    hOldWnd% = GetFocus()
      'Set the focus to the list box.
    List1.SetFocus
      'Get the handle to the list box.
    lbhWnd% = GetFocus()
      'Set up the array of defined tab stops.
    tabs(1) = 10
    tabs(2) = 50
    tabs(3) = 90
      'Send a message to the message queue.
    retVal& = SendMessage(lbhWnd%, LB_SETTABSTOPS, 3, tabs(1))
      'Restore the handle to whoever had it.
    R% = APISetFocus(hOldWnd%)

      'Place some elements into the list box.
    List1.AddItem "Name" + Chr$(9) + "Rank" + Chr$(9) + "Serial#"
    List1.AddItem "J. Doe" + Chr$(9) + "O-3" + Chr$(9) + "1234"
    List1.AddItem "J. Blow" + Chr$(9) + "E-1" + Chr$(9) + "5678"
    List1.AddItem "F. Smith" + Chr$(9) + "O-6" + Chr$(9) + "0192"
End Sub
```

References

How to Clear a VB List Box with a Windows API Function

Article Number: Q71069

Summary:

Customers commonly ask how to quickly clear the contents of a list box without clearing one item at a time. The following article shows how to instantly clear the contents of a list box by sending the list box a LB_RESETCONTENT message.

More Information:

There is no single command within Visual Basic that will clear out the entries of a list box. However, by using the **SendMessage**() function, you can clear the list box with one command. The arguments to **SendMessage**() with the LB_RESETCONTENT parameter are:

SendMessage(hWnd%, wMsg%, wParam%, lParam&)

Parameter	Type/Description
hWnd%	Identifies the window that is to receive the message.
wMsg%	The message to be sent. (&H405)
wParam%	Is not used. (NULL)
lParam&	Is not used. (NULL)

Specifying wMsg% equal to &H405 removes all strings from the list box and frees any memory allocated for those strings.

To get hWnd% you must call the Windows API function **GetFocus**(). This method will return the handle to the control that currently has focus; in this case the list box that you want to delete all items from.

The code listed below demonstrates how to delete entries from a list box:

1. Create a list box called List1 on Form1.

2. Declare the following Windows API functions at the module level or in the Global section of your code as follows:

```
Declare Function SendMessage% Lib "user" (
        ByVal hWnd%,
        ByVal wMsg%,
        ByVal wParam%,
        ByVal lParam&)
Declare Function GetFocus% Lib "user" ()
Declare Function APISetFocus% Lib "user" Alias "SetFocus" (ByVal hWnd%)
```

Note: Each Declare statement must be written on one line

3. Declare the following constants in the same section:

```
Const WM_USER = &H400
Const LB_RESETCONTENT = WM_USER + 5
```

4. Create a Sub within the (declarations) section of the code window with the following code:

```
Sub ClearListBox (Ctrl As Control)
    hWndOld% = GetFocus()
    Ctrl.SetFocus
    x = SendMessage(GetFocus(), LB_RESETCONTENT, 0, 0)
    Suc% = APISetFocus(hWndOld%)
End Sub
```

5. Within some event procedure, call ClearListBox with the name of the list box as a parameter:

```
Sub Form_Click ()
    ClearListBox List1
End Sub
```

6. Place some entries into the list box:

```
Sub Form_Load ()
    For i = 1 To 10
      'Put something into list box.
        List1.AddItem Format$(i)
    Next
End Sub
```

7. Run the program and click anywhere on Form1. This will clear the list box.

References

How to Flood Fill in VB Using Windows API Function Call

Article Number: 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 **ExtFloodFill**() to achieve the desired effect.

More Information:

The Windows API function call **ExtFloodFill**() fills an area of the display surface with the current brush, as shown in the example below.

Code Example

From VB.EXE's 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%)
```

To demonstrate several fill examples, create a picture box called Picture1. Set the following properties:

```
        'Scale picture to size of imported picture.
AutoSize = TRUE
        'This will be the selected fill color.
FillColor = &HFF00FF
        'Necessary to create a fill pattern.
FillStyle = Solid
        'This should be in your Windows directory.
Picture = Chess.bmp
```

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.
                Fill until crColor& encountered.
    Const FLOODFILLBORDER = 0
      'Fill surface until crColor& not encountered.
    Const FLOODFILLSURFACE = 1
    X% = 1
    Y% = 1
    crColor& = RGB(0, 0, 0)
    wFillType% = FLOODFILLSURFACE
    Suc% = ExtFloodFill(picture1.hDC, X%, Y%, crColor&, wFillType%)
End Sub
```

When you click 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:

```
        'Color to look for.
```

```
crColor& = RGB(255, 0, 0)
wFillType% = FLOODFILLBORDER
Suc% =   ExtFloodFill(picture1.hDC, X%, Y%, crColor&, wFillType%)
```

Clicking 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."

References

How to Use Windows 3.0 BitBlt Function from Visual Basic

Article Number: Q71104

Summary:

The Windows GDI.DLL has a function call **BitBlt**(), which will move the source device given by the hSrcDC parameter to the destination device given by the hDestDC parameter. This article explains in detail the arguments of the Windows **BitBlt**() function call.

More Information:

To use **BitBlt**() within a Visual Basic application, you must Declare the **BitBlt**() function in either the Global section or within the (declaration) section of your Code window. Declare the Function as follows:

```
Declare Function BitBlt%Lib "gdi" (
        ByVal hDestDC%,
        ByVal X%, ByVal Y%,
        ByVal nWidth%,
        ByVal nHeight%,
        ByVal hSrcDC%,
        ByVal XSrc%,
        ByVal YSrc%,
        ByVal dwRop&)
```

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

The following formal parameters are defined as:

Parameter	Type/Description
hDestDC%	Specifies the device context that is to receive the bitmap.
X,%,Y%	Specifies the logical x-coordinate and y-coordinate of the upper-left corner of the destination rectangle.
nWidth%	Specifies the width (in logical units) of the destination rectangle and the source bitmap.
nHeight%	Specifies the height (in logical units) of the destination rectangle and the source bitmap.
hSrcDC%	Identifies the device context from which the bitmap will be copied. It must be NULL (zero) if the dwRop& parameter specifies a raster operation that does not include a source.
XSrc%	Specifies the logical x-coordinate and the y-coordinate of the upper-left corner of the source bitmap.
dwRop&	Specifies the raster operation to be performed as defined below.

The following raster operations are defined using the predefined constants found in the WINDOWS.H file supplied with the Microsoft Windows version 3.0 Software Development Kit (SDK). The value within the "()" is the value to assign to the dwRop& variable.

Code/Value (Hex)	Result
BLACKNESS (42)	Turns output black.

DSINVERT(550009)	Inverts the destination bitmap.
MERGECOPY(C000CA)	Combines the pattern and the source bitmap using the Boolean AND operation.
MERGEPAINT(BB0226)	Combines the inverted source bitmap with the destination bitmap using the Boolean OR operator.
NOTSRCCOPY(330008)	Copies the inverted source bitmap to the destination.
NOTSRCERASE(1100A6)	Inverts the result of combining the destination and source bitmaps using the Boolean OR operator.
PATCOPY(F00021)	Copies the pattern to the destination bitmap.
PATINVERT(5A0049)	Combines the destination bitmap with the pattern using the Boolean XOR operator.
PATPAINT(FB0A09)	Combines the inverted source bitmap with the pattern using the Boolean OR operator. Combines the result of this operation with the destination bitmap using the Boolean OR operator.
SRCAND(8800C6)	Combines pixels of the destination and source bitmaps using the Boolean AND operator.
SRCCOPY(CC0020)	Copies the source bitmap to the destination bitmap.
SRCERASE(4400328)	Inverts the destination bitmap and combines the results with the source bitmap using the Boolean AND operator.
SRCINVERT(660046)	Combines pixels of the destination and source bitmaps using the Boolean XOR operator.
SRCPAINT(EE0086)	Combines pixels of the destination and source bitmaps using the Boolean OR operator.
WHITENESS(FF0062)	Turns all output white.

Below is an example of how to copy the contents of a picture control to the contents of another picture control.

Define a form with two picture controls. Display some graphics on Picture1 by loading them from a picture file or pasting them from the Clipboard at design time. You can load a picture from a file as follows: from the Properties bar, select Picture from the Properties list box and click the arrow at the right of the Settings box, then select the desired picture file (such as a .BMP or .ICO file supplied with Microsoft Windows) from the dialog box.

Add the following code to the Form_Click procedure. Run the program and click the form. The contents of the first picture will be displayed in place of the second picture.

```
Sub Form_Click ()
    'Assign information of the destination bitmap. Note that BitBlt() requires coordinates in
        pixels.
    Const PIXEL = 3
    Picture1.ScaleMode = PIXEL
```

```
        Picture2.ScaleMode = PIXEL
        hDestDC% = Picture2.hDC
        X% = 0: Y% = 0
        nWidth% = Picture2.ScaleWidth
        nHeight% = Picture2.ScaleHeight
          'Assign information of the source bitmap.
        hSrcDC% = Picture1.hDC
        XSrc% = 0: YSrc% = 0
          'Assign the SRCCOPY constant to the raster operation.
        dwRop& = &HCC0020
        Suc% = BitBlt (hDestDC%, X%, Y%, nWidth%, nHeight%, hSrcDC%, XSrc%, YSrc%,
                dwRop&)
    End Sub
```

References

How to Implement Bitmaps Within Visual Basic Menus

Article Number: Q71281

Summary:

There is no Visual Basic command for adding bitmaps to the menu system. However, there are several Windows API functions that you can call that will place bitmaps within the menu system. You can also change the default check mark that is displayed.

More Information:

There are several Windows API functions that you can call that will display bitmaps instead of text in the menu system.

Below is a list of the required Windows API functions:

GetMenu% (hWnd%)

Parameter	Type/Description
hWnd%	Identifies the window whose menu is to be examined.
Returns	Handle to the menu.

GetSubMenu% (hMenu%, nPos%)

Parameter	Type/Description
hMenu%	Identifies the menu.
nPos%	Specifies the position (zero-based) in the given submenu of the pop-up menu.
Returns	Handle to the given pop-up menu.

GetMenuItemID% (hMenu%, nPos%)

Parameter	Type/Description
hMenu%	Identifies the handle to the pop-up menu that contains the item whose ID is being retrieved.
nPos%	Specifies the position (zero-based) of the menu whose ID is being retrieved.
Returns	The item ID for the specified item in the pop-up menu.

ModifyMenu% (hMenu%, nPos%, wFlags%, wIDNewItem%, lpNewItem&)

Parameter	Type/Description
hMenu%	Identifies the handle to the pop-up menu that contains the item whose ID is being retrieved.
nPos%	Specifies the menu item to be changed. The interpretation of the nPos parameter depends on the wFlags parameter.
wFlags%	BF_BITMAP = &H4
wIDNewItem%	Specifies the command ID of the modified menu item.
lpNewItem&	32-bit handle to the bitmap.

Returns	TRUE if successful, FALSE if unsuccessful.

SetMenuItemBitmaps% (hMenu%, nPos%, wFlags%, hBitmapUnchecked%, hBitmapChecked%)

Parameter	Type/Description
hMenu%	Identifies menu to be changed.
nPos%	Command ID of the menu item.
wFlags%	&H0
hBitmapUnchecked%	Handle to "unchecked" bitmap.
hBitmapChecked%	Handle to the "check" bitmap.
Returns	TRUE if successful, FALSE if unsuccessful.

There are two different ways to implement bitmaps in Visual Basic. The first method is using static bitmaps. The other method is using dynamic bitmaps.

A static bitmap is fixed and does not change during the execution of your program (such as when it is taken from an unchanging .BMP file). A dynamic bitmap changes during execution of your program. You may change dynamic bitmap attributes such as color, size, and text. The sample code below describes how to perform both types of menus.

Define a menu system using the Menu Design window. Create a menu system such as the following:

Caption	CtlName	Indented	Index
BitMenu	TopMenu	No	
Sub Menu0	SubMenu	Once	0
Sub Menu1	SubMenu	Once	1
Sub Menu2	SubMenu	Once	2

Create a Picture Control Array with three bitmaps. This can be accomplished by creating three picture controls with the same CtlName using the Properties list box.

Caption	CtlName	Index	FontSize
Picture0	Picture1	0	Set different
Picture1	Picture1	1	font sizes for
Picture2	Picture1	2	each of the pictures

Both types of bitmap implementations will need to have the following declarations in the declaration or global section of your code:

```
Function GetMenu% Lib "user" (ByVal hwnd%)

Declare Function GetSubMenu% Lib "user" (
        ByVal hMenu%,
        ByVal nPos%)

Declare Function GetMenuItemID% Lib "user" (
        ByVal hMenu%,
        ByVal, nPos%)

Declare Function ModifyMenu% Lib "user" (
        ByVal hMenu%,
```

```
            ByVal nPosition%,
            ByVal wFlags%,
            ByVal wIDNewItem%,
            ByVal lpNewItem&)

Declare Function SetMenuItemBitmaps% Lib "user" (
            ByVal hMenu%,
            ByVal nPosition%,
            ByVal wFlags%,
            ByVal hBitmapUnchecked%,
            ByVal BitmapChecked%)

      'NOTE: Each Declare statement above must be on just one line.
Const MF_BITMAP = &H4
      'Defined for dynamic bitmaps only.
Const CLR_MENUBAR = &H80000004
Const TRUE = -1, FALSE = 0
Const Number_of_Menu_Selections = 3
```

The following Sub will also need to be defined to handle the actual redefinition of the "check" bitmap:

```
Sub SubMenu_Click (Index As Integer)
      'Uncheck presently checked item, check new item, store index
    Static LastSelection%
    SubMenu(LastSelection%).Checked = FALSE
    SubMenu(Index).Checked = TRUE
    LastSelection% = Index
End Sub
```

The following examples show the two ways to implement the dynamic and static menu bitmaps.

Static Bitmap Menus

Add the code listed below to the appropriate Sub:

```
Sub Form_Load ()
      'Get the handle to the top level Menu
    hMenu% = GetMenu(hwnd)
      'Get the handle to the SubMenu of top level Menu
    hSubMenu% = GetSubMenu(hMenu%, 0)
    For I% = 0 To Number_of_Menu_Selections - 1
      'get the handle to the specific item in SubMenu
        menuID% = GetMenuItemID(hSubMenu%, I%)
      'Place the bitmap into the I'th submenu location
        X% = ModifyMenu(hMenu%, menuID%, MF_BITMAP, menuID%,
                CLng(picture1(I%).Picture))
      'Assign bitmap for the check mark to the I'th submenu
        X% = SetMenuItemBitmaps(hMenu%, menuID%, 0, 0,CLng(picture2.Picture))
    Next I%
End Sub
```

Dynamic Bitmap Menus

This code sample will change the actual menu bitmap's size, font size, color, and caption. Run the

application and select the BitMenu and view the selections. Then click on the form and revisit the BitMenu.

```
Sub Form_Click ()
    For i% = 0 To Number_of_Menu_Selections
      'Place some text into the menu.
        SubMenu(i%).Caption = Picture1(i%).FontName +Str$(Picture1(i%).FontSize) + " Pnt"
      '1. Must be AutoRedraw for Image().
      '2. Set Backcolor of Picture control to that of the current system Menu Bar color, so dynamic
            bitmaps will appear as normal menu items when menu bar color is changed in the
            Control Panel.
      '3. See the bitmaps on screen; this could all be done at design time.
        Picture1(i%).AutoRedraw = TRUE
        Picture1(i%).BackColor = CLR_MENUBAR
        Picture1(i%).Visible = FALSE
      'Set the width and height of the Picture controls based on their corresponding menu item's
            caption, and the picture control's Font and FontSize. DoEvents() is necessary to
            make new dimension values take effect prior to exiting this Sub.
        Picture1(i%).Width = Picture1(i%).TextWidth(SubMenu(i%).Caption)
        Picture1(i%).Height = Picture1(i%).TextHeight(SubMenu(i%).Caption)
        Picture1(i%).Print SubMenu(i%).Caption
      'Set picture control's backgroup picture (bitmap) to its image.
        Picture1(i%).Picture = Picture1(i%).Image
        X% = DoEvents()
    Next i%
      'Get handle to form's menu.
    hMenu% = GetMenu(Form1.hWnd)
      'Get handle to the specific menu in top-level menu.
    hSubMenu% = GetSubMenu(hMenu%, 0)
    For i% = 0 To Number_of_Menu_Selections
      'Get ID of submenu.
        menuID% = GetMenuItemID(hSubMenu%, i%)
      'Replace menu text with bitmap from corresponding picture control.
        X% = ModifyMenu(hMenu%, menuID%, MF_BITMAP, menuID%,
              CLng(Picture1(i%).Picture))
      'Replace bitmap for menu check mark with custom check bitmap.
        X% = SetMenuItemBitmaps(hMenu%, menuID%, 0, 0,CLng(picture2.Picture))
    Next i%
End Sub
```

References

ExtFloodFill Won't Fill Over QBColors (1-8) If AutoRedraw = True

Article Number: Q75640

Summary:

There is a problem in the Windows 3.0 environment with the **ExtFloodFill**() API function. If you try to use the **ExtFloodFill**() API along with the QBColor function that is included in Microsoft Visual Basic version 1.0, the first eight colors are displayed incorrectly.

This problem causes the Fill Tool of the IconWorks sample application (provided with Microsoft Visual Basic programming system version 1.0 for Windows) to fail when attempting to fill over QBColors (1-8).

More Information:

Steps to Reproduce Problem

1. Start Visual Basic with a New Project.

2. Place a picture box on the Form. From the Properties bar, set the AutoRedraw equal to TRUE and the FillStyle equal to Solid for the picture box.

3. Place the following code in the General Declarations section of the code window for Form1:

```
DefInt A-Z
Declare Function ExtFloodFill% Lib "GDI" (
        ByVal hdc,
        ByVal x,
        ByVal y,
        ByVal crcolor as Long,
        ByVal wfilltype)
```

4. Place the following code inside the Form_Click event procedure.

```
Sub Form_Click ()
    Static I
    I= I + 1
    Picture1.BackColor = QBColor(I)
    x = ExtFloodFill(Picture1.hdc, 1, 1, Picture1.BackColor, 1)
    Print I;x
    Picture1.Refresh
End Sub
```

5. Run the sample by pressing the F5 key. You should see that various colors are incorrectly displayed for QBColors 1-8 and that the return value from **ExtFloodFill**(), held in x, is 0. QBColors 1-8 should be displaying black and the value for x should equal 1, not 0. QBColors 9-15 are correctly displayed.

References

How to Print a VB Picture Control Using Windows API Functions

Article Number: Q77060

Summary:

This article explains how to print Visual Basic picture control using several Windows 3.0 API function calls.

More Information:

To print a picture control from Visual Basic, you must use the .PrintForm method. Although printing a picture control can very useful, there is no way to print just a picture control without the use of API function calls. Calling API functions to print a picture control is useful when you want to control the location or size of the printed image. Calling API functions to print a picture control is also useful if you want to include other images or text along with the picture image on a single sheet of paper.

To print a bitmap, you need to do the following:

1. Create a memory device context that is compatible with the bitmap [**CreateCompatibleDC**()]. A memory device context is a block of memory that represents a display surface. It is used to prepare images before copying them to the actual device surface of the compatible device.

2. Save the current object [**SelectObject**()] and select the picture control using the handle from the memory device context.

3. Use **BitBlt()** or **StretchBlt**() to copy the bitmap from the memory device context to the printer.

4. Remove the bitmap from the memory device context [**SelectObject**()] and delete the device context [**DeleteDC**()].

The following steps demonstrate this process:

1. Create a new form.

2. Add a picture (Picture1) control.

3. Add a command (Command1) button.

4. Display some graphics in Picture1 by loading them from a picture file or pasting them from the Clipboard at design time. You can load a picture from a file as follows:

   a. On the Properties bar, select Picture from the Properties list box.

   b. Click the arrow at the right of the Settings box, then select the desired picture file (such as a .BMP or .ICO file supplied with Microsoft Windows) from the dialog box.

5. Add the following declarations to the global declaration section of the Code window:

```
Declare Function CreateCompatibleDC% Lib "GDI" (ByVal hDC%)
Declare Function SelectObject% Lib "GDI" (ByVal hDC%, ByVal hObject%)
Declare Function StretchBlt% Lib "GDI" (
        ByVal hDC%, ByVal X%,
        ByVal Y%, ByVal nWidth%,
        ByVal nHght%, ByVal hSrcDC%,
        ByVal XSrc%, ByVal YSrc%,
```

```
            ByVal nSrcWidth%,
            ByVal nSrcHeight%,
            ByVal dwRop&)
Declare Function DeleteDC% Lib "GDI" (ByVal hDC%)
Declare Function Escape% Lib "GDI" (
            ByVal hDC As Integer,
            ByVal nEscape As Integer,
            ByVal nCount As Integer,
            lpInData As Any,
            lpOutData As Any)
```

Note: Each Declare statement must be on one line.

6. Add the following code to the Command_Click event:

```
Sub Command1_Click ()
    Const NULL = 0&
    Const SRCCOPY = &HCC0020
    Const NEWFRAME = 1
    Const PIXEL = 3
      'display hour glass
    MousePointer = 11
      'StretchBlt requires pixel coordinates.
    Picture1.ScaleMode = PIXEL
    Printer.ScaleMode = PIXEL
    Printer.Print " "
    hMemoryDC% = CreateCompatibleDC(Picture1.hDC)
    hOldBitMap% = SelectObject(hMemoryDC%, Picture1.Picture)
    ApiError% = StretchBlt(Printer.hDC, 0, 0, Printer.ScaleWidth, Printer.ScaleHeight,
            hMemoryDC%, 0, 0, Picture1.ScaleWidth, Picture1.ScaleHeight, SRCCOPY)
    hOldBitMap% = SelectObject(hMemoryDC%, hOldBitMap%)
    ApiError% = DeleteDC(hMemoryDC%)
    Print Escape(Printer.hDC, NEWFRAME, 0, NULL, NULL)
    Printer.EndDoc
    MousePointer = 1
End Sub
```

7. Run the program to copy the bitmap to the printer.

Warning: If you have selected a low resolution from the Print Manager, printing the bitmap will proceed quicker (the lower the resolution, the faster the print time). Some printers will also print the image faster than others Using an HP LaserJet II, this process proceeds at an acceptable speed. While designing your software, you may want to keep the bitmap printing at the lowest possible resolution. The print resolution can be changed from the Windows Control Panel.

References

How to Create Flashing/Rotating Rubber-Band Box in VB

Article Number: Q71489

Summary:

Several programs, such as Microsoft Excel for Windows, create a flashing border (which appears to rotate) when items are selected using the Copy command on the Edit menu. You can create a flashing, rotating border with the DrawMode and DrawStyle properties of a Visual Basic form.

More Information:

By drawing a dashed line on the form and then within a timer event creating a solid line on the dashed line with DrawMode set to INVERSE, you can create the effect of a flashing border that appears to rotate.

You can draw a rotating rubber-band box as follows:

1. Draw a line using: DrawStyle = 2 {Dot}

2. Save the [form].DrawMode and the [form].DrawStyle.

3. Set the [form].DrawMode = 6 {Inverse}.

4. Set [form].DrawStyle = 0 {Solid}.

5. Draw the same line as in step 1.

6. Reset the properties saved in step 2.

7. Delay some time interval.

8. Repeat starting at step 2.

The following code demonstrates the rotating (flashing) border. Pressing the mouse button and then dragging the cursor some distance will create a dotted line. Releasing the button will display a rotating rubber-band box.

In VB.EXE, create a form called Form1. On Form1, create a timer control with the name Timer1 and with an interval of 100.

Duplicate the following code within the general declaration section of your Code window:

```
   'Characteristic of DrawStyle property(Inverse).
Const INVERSE = 6
   'Characteristic of DrawMode property.
Const SOLID = 0
   'Characteristic of DrawMode property.
Const DOT = 2
Const TRUE = -1
Const FALSE = 0
Dim OldX, OldY, StartX, StartY As Single
```

Add the following code in the appropriate event procedures for Form1:

```
Sub Form_Load ()
```

```
        'Must draw a dotted line to create effect.   Loading a bitmap. Not required but shows   full
                extent of line drawing.
    DrawStyle = DOT
End Sub
```

```
Sub Timer1_Timer ()
    SavedDrawStyle% = DrawStyle
      'Solid is need to create the inverse of the dashed line.
    DrawStyle = SOLID
      'Invert the dashed line.
    Call DrawLine(StartX, StartY, OldX, OldY)
      'Restore the DrawStyle back to what it was previously.
    DrawStyle = SavedDrawStyle%
End Sub
```

```
Sub Form_MouseDown (Button As Integer, Shift As Integer, X As Single, Y As Single)
        'The above Sub statement must be on just one line. Don't add effect as you draw box.
    Timer1.Enabled = FALSE
      Save the start locations.
    StartX = X
    StartY = Y
      'Set the last coord. to start locations.
    OldX = StartX
    OldY = StartY
End Sub
```

```
Sub Form_MouseMove (Button As Integer, Shift As Integer, X As Single, Y As Single)
        'The above Sub statement must be on just one line. If button is depress then...
    If Button Then
      'Restore previous lines background.
        Call DrawLine(StartX, StartY, OldX, OldY)
      'Draw new line.
        Call DrawLine(StartX, StartY, X, Y)
      'Save coordinates for next call.
        OldX = X : OldY = Y
    End If
End Sub
```

```
Sub DrawLine (X1, Y1, X2, Y2 As Single)
        'Save the current mode so that you can reset it on exit from this sub routine. Not needed in
                the sample but would need it if you are not sure what the DrawMode was on entry to
                this procedure.
    SavedMode% = DrawMode
      'Set to XOR
    DrawMode = INVERSE
      'Draw a box
    Line (X1, Y1)-(X2, Y2), , B
      'Reset the DrawMode
    DrawMode = SavedMode%
End Sub
```

```
Sub Form_MouseUp (Button As Integer, Shift As Integer, X As Single, Y As Single)
        'The above Sub statement must be on just one line.
        StartEvent = FALSE
        Timer1.Enabled = TRUE
```

```
End Sub
```

How to Create Rubber-Band Lines/Boxes in Visual Basic

Article Number: Q71488

Summary:

Creating rubber-band lines or boxes within Visual Basic can be done using the DrawMode property. Rubber bands are lines that stretch as you move the mouse cursor from a specified point to a new location. This can be very useful in graphics programs and when defining sections of the screen for clipping routines.

More Information:

The theory of drawing a rubber-band box is as follows:

1. Draw a line from the initial point to the location of the mouse cursor using:[form].DrawMode = 6. {INVERT}

2. Move the mouse cursor.

3. Save the DrawMode.

4. Set the [form].DrawMode to 6. {INVERT}

5. Draw the same line that was drawn in step 1. This will restore the image underneath the line.

6. Set the [form].DrawMode back to the initial DrawMode saved in step 3.

7. Repeat the cycle again.

DrawMode equal to INVERT allows the line to be created using the inverse of the background color. This allows the line to be displayed on all colors.

The sample below demonstrates the rubber-band line and the rubber-band box. Clicking the command buttons allows the user to select either a rubber-band line or a rubber-band box. The user can also select a solid line or a dashed line.

Create and set the following controls and properties:

CtlName	Caption	Picture
Form1	Form1	C:\WINDOWS\CHESS.BMP
Command1	RubberBand	
Command2	RubberBox	
Command3	Dotted	
Command4	Solid	

In the general section of your code, define the following constants:

```
   'Characteristic of DrawMode property(XOR).
Const INVERSE = 6
   'Characteristic of DrawStyle property.
Const SOLID = 0
   'Characteristic of DrawStyle property.
Const DOT = 2
Const TRUE = -1
Const FALSE = 0
```

```
    'Boolean-whether drawing box or line
  Dim DrawBox As Integer
    'Mouse locations
  Dim OldX, OldY, StartX, StartY As Single
```

In the appropriate procedures, add the following code:

```
Sub Form_MouseDown (Button As Integer, Shift As Integer, X As Single, Y As Single)
    'Store the initial start of the line to draw.
  StartX = X
  StartY = Y
    'Make the last location equal the starting location
  OldX = StartX
  OldY = StartY
End Sub
```

```
Sub Form_MouseMove (Button As Integer, Shift As Integer, X As Single, Y As Single)
    'If the button is pressed then...
  If Button Then
    'Erase the previous line.
      Call DrawLine(StartX, StartY, OldX, OldY)
    'Draw the new line.
      Call DrawLine(StartX, StartY, X, Y)
    'Save the coordinates for the next call.
  OldX = X
  OldY = Y
  End If
End Sub
```

```
Sub DrawLine (X1, Y1, X2, Y2 As Single)
    'Save the current mode so that you can reset it on exit from this subroutine. Not needed in
          the sample but would need it if you are not sure what the DrawMode was on entry to
          this procedure.
  SavedMode% = DrawMode
    'Set to XOR
  DrawMode = INVERSE
    'Draw a box or line
  If DrawBox Then
      Line (X1, Y1)-(X2, Y2), , B
  Else
      Line (X1, Y1)-(X2, Y2)
  End If
    'Reset the DrawMode
  DrawMode = SavedMode%
End Sub
```

```
Sub Form_MouseUp (Button As Integer, Shift As Integer, X As Single, Y As Single)
    'Stop drawing lines/boxes.
  StartEvent = FALSE
End Sub
```

```
Sub Command2_Click ()
    'Boolean value to determine whether to draw a line or box.
  DrawBox = TRUE
End Sub
```

```
Sub Command1_Click ()
        'Boolean value to determine whether to draw a line or box.
    DrawBox = FALSE
End Sub
```

```
Sub Command3_Click ()
        'Create a dotted line
    Form1.DrawStyle = DOT
End Sub
```

```
Sub Command4_Click ()
        'Create a solid line.
    Form1.DrawStyle = SOLID
End Sub
```

How to Make a VB Text Box Control with a Password (*) Style

Article Number: Q71457

Summary:

In the password style for a control, each character is displayed as an asterisk (*) as it is typed into the control. Microsoft Visual Basic does not have any intrinsic function for creating an edit control with the password style. You can add the password style to a Visual Basic text box control by using several Windows API function calls.

More Information:

Adding the password style to a text box control can be performed by calling several Windows API function calls. These functions are **GetFocus**(), **GetWindowLong**(), **SetWindowLong**(), and **SendMessage**().

The function **GetFocus**() requires no parameters. This function returns an integer value that represents the handle to the control. Use **GetFocus**() to get a handle to the control that currently has the focus.

```
hWd% = GetFocus()
```

After you have received the handle to the text box control, you can call **GetWindowLong**() to retrieve the style flags for the control by using the second parameter of GWL_STYLE. Once you have the style flags, you can use the bitwise OR operation to set the ES_PASSWORD style.

```
StyleFlags& = GetWindowLong (hWd%, GWL_STYLE)
StyleFlags& = StyleFlags& OR ES_PASSWORD
```

After adding the password style to the existing style for the control, you can call **SetWindowLong**() to change the style associated to the control. You would call **SendMessage**() with the message parameter of ES_SETPASSWORDCHAR to inform the control what character is to be used as the password mask; the default character is an asterisk (*).

```
StyleFlags& = SetWindowLong (hWd%, GWL_STYLE, StyleFlags&)
PasswordMask% = Asc("*")
X = SendMessage (hWd%, ES_SETPASSWORDCHAR, PasswordMask%, 0&)
```

To create a text box with the password style in Visual Basic, create a text box with the name Text1 on Form1.

Declare the following Windows API functions and CONST variables in the Global section of your code:

```
Declare Function GetFocus% Lib "User" () As Integer

Declare Function GetWindowLong& Lib "User" (
        ByVal hWd%,
        ByVal nIndex%)

Declare Function SetWindowLong& Lib "User" (
        ByVal hWd%,
        ByVal nIndex%,
        ByVal dwNewLong&)
```

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

Global Const WM_USER = &H400
Global Const EM_SETPASSWORDCHAR = WM_USER + 28
Global Const ES_PASSWORD = &H20
Global Const GWL_STYLE = -16
```

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

Include the following code in the (general) section of the Form1 module:

```
Sub Make_Password_Control (Flag As Integer, PasswordMask As Integer)
      'Window Handle for the control
    Dim hWd As Integer
       'Window Style for the control
    Dim StyleFl As Long
    If Not Flag Then
      'The control should have a STATIC flag to minimize the execution of the code.   We don't
            want to perform this code all the time, just once.
        Flag = Not Flag
        hWd = GetFocus()
        StyleFl = GetWindowLong(hWd, GWL_STYLE)
        StyleFl = StyleFlags Or ES_PASSWORD
        StyleFl = SetWindowLong(hWd, GWL_STYLE, StyleFlags)
        StyleFl = SendMessage(hWd, EM_SETPASSWORDCHAR, PasswordMask, 0&)
    End If
End Sub
```

Include the following code in the Text1_GotFocus event in the Form1 module:

```
Sub Text1_GotFocus ()
    Static Already_Password As Integer
    Make_Password_Control Already_Password, Asc("*")
End Sub
```

References

How to Set Character Limit in VB Combo/Text Box; SendMessage API

Article Number: Q72677

Summary:

You can specify a limit to the amount of text that can be entered into a text and/or combo box by calling **SendMessage**() (a Windows 3.0 API function) with the EM_LIMITTEXT constant.

More Information:

The constant EM_LIMITTEXT can be sent to the **SendMessage**() Windows 3.0 API function to limit the length of a string entered into a combo and/or text box.

Another method is to 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 (0).

A cleaner way 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)

Parameter	Type/Description
hWnd%	Handle to the text box.
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	Not used.

As an example, do the following:

1. Create a form called Form1.

2. Add to Form1 a combo box called Combo1.

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

```
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
```

Note: Each Declare statement must be on just one line.

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

```
Sub Form_Load ()
        'Must show form to work on it
    Form1.Show
        'Set the focus to the list box
    Combo1.SetFocus
        'Get the handle to the list box
    cbhWnd% = GetFocus()
        'Specify the largest string
    TextLimit% = 5
    retVal = SendMessage(cbhWnd%, EM_LIMITTEXT, TextLimit%, 0)
End Sub
```

5. Run the program and enter some text into the combo box. Note that you are able to enter only five characters into the combo box.

References

Determining Number of Lines in VB Text Box; SendMessage API

Article Number: 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.

SendMessage& (hWd%, wMsg%, wParam%, lParam%)

Parameter	Type/Description
hWd%	Handle to the text box.
wMsg%	EM_GETLINECOUNT(&H40A)
wParam%	0
lParam%	0

More Information:

To determine the number of lines within a text box:

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.

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

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

3. You will need to declare another API routine to get the handle of the text box. Declare this also in your global declarations section of your Code window. The returned value will become the hWd% argument to the **SendMessage**() function.

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

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

```
Sub Command1_Click ()
    'defined within SDK WINDOWS.H
  Const EM_GETLINECOUNT = &H40A
    'command button has focus, give focus to text box.
  Text1.SetFocus
    'get the handle of the text box.
  hWd% = GetFocus()
    'print the amount of lines to the immediate window.
  Debug.Print SendMessage(hWd%, EM_GETLINECOUNT, 0, 0)
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 to the immediate window.

References

How to Scroll VB Text Box Programatically and Specify Lines

Article Number: 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 Visual Basic text box. By using **SendMessage**(), you can also scroll text programatically, without user interaction. This technique extends Visual Basic's 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 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 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 as follows:

SendMessage& (hWd%, EM_LINESCROLL, wParam%, lParam%, lParam&)

Parameter	Type/Description
hWd%	Handle to the text box.
wMsg%	EM_GETLINECOUNT(&H40A)
wParam%	0
lParam%	0
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" ()

Sub Command_Scroll_Click ()
        'Store the window handle of the control that currently has the focus.
    OldhWnd% = GetFocus ()
    Text1.SetFocus
    hWd% = GetFocus()
```

```
End Sub
```

To scroll text horizontally, the text box must have a horizontal scroll bar, and 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 the 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 (ALT+F+N).

2. Double-click Form1 to bring up the Code window.

3. Add the following API declaration in 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 **APISetFocus**() because a SetFocus method already exists within Visual Basic.

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

```
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% = GetFocus%()
      'set focus to the passed control (Text control)
    TextBox.SetFocus
      'get the handle to current focus (Text control)
    TextWnd% = GetFocus%()
      'scroll the lines
    Success& = SendMessage(TextWnd%, EM_LINESCROLL, 0, Lines&)
      'restore the focus to the original control, Command1 or Command2
    r% = APISetFocus% (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.

References

Using Windows API Functions to Better Manipulate Text Boxes

Article Number: Q76518

Summary:

By calling Windows API functions from Visual Basic, you can retrieve text box (or edit control) information that you cannot obtain using only Visual Basic's built-in features.

This article supplies a sample program that performs the following useful features (making use of the Windows message constants shown in parentheses, obtained by calling Windows API routines):

-- Copy a specific line of text from the text box (EM_GETLINE).

-- Retrieve the number of lines within the text box (EM_GETLINECOUNT).

-- Position the cursor at a specific character location (EM_GETSEL) in the text box.

-- Retrieve the line number of a specific character location in the text box (EM_LINEFROMCHAR).

-- Retrieve the number of lines prior to a specified character position in the text box (EM_LINEINDEX).

-- Retrieve the number of characters in a specified line in the text box (EM_LINELENGTH).

-- Replace specified text with another text string (EM_REPLACESEL).

For a separate article that explains how to specify the amount of text allowable within a text control, search on the following word:

EM_LIMITTEXT

More Information:

The Windows API file USER.EXE defines the **SendMessage**() function that will return or perform a specific event on your edit control. To create an example that will display specific information about your edit control, do the following:

1. Create a form called Form1 with the following child controls and properties:

Control	CtlName	Height	Left	Top	Width
Label	aGetLine		360	120	
Label	aGetLineCount		360	480	
Label	aGetSel		360	840	
Label	aLineFromChar		360	1200	
Label	aLineIndex		360	560	
Label	aLineLenght		360	1920	
Label	aReplaceSel		360	2280	
Command	Command1	375	360	2640	1815
Text	Text1	1815	2640	480	3495
Text	Text2	375	2520	2640	3615

2. Set each label's AutoSize property to TRUE.

3. Set the Text1.MultiLine property to TRUE.

4. Change the Text2.Caption to "Insert this text --->".

5. Within the Global Declaration section, add the following code:

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

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

6. After adding the code listed below to your form, run the program. Whenever a key is released, the labels will be updated with the new information about your text box.

```
Sub Form_Load ()
    Show
      'used to display the correct text.
    X% = fReplaceSel("")
End Sub
```

```
Sub Text1_KeyUp (KeyCode As Integer, Shift As Integer)
      'Update the text control information whenever the key is pressed and released.
    CharPos& = fGetSel()
    LineNumber& = fLineFromChar(CharPos&)
    X% = fGetLine(LineNumber&)
    X% = fGetLineCount()
    X% = fLineIndex(LineNumber&)
    X% = fLineLength(CharPos&)
End Sub
```

```
Sub Command1_Click ()
      'This routine will insert a line of text at the current location of the caret.
    D$ = Text2.text
    CharPos% = fGetSel()
    X% = fReplaceSel(D$)
    X% = fSetSel(CharPos%)

      'Text has been inserted at the caret location.   Now update the text controls information.
    Call Text1_KeyUp(0, 0)
    Text1.SetFocus
End Sub
```

```
Function fGetLineCount& ()
       'This function will return the number of lines in the edit control.
    Const EM_GETLINECOUNT = &H400 + 10

    Text1.SetFocus
    Pos% = SendMessage(GetFocus(), EM_GETLINECOUNT, 0&, 0&)
    aGetLineCount.Caption = "GetLineCount = " + Str$(Pos%)
    fGetLineCount = Pos%
End Function
```

```
Function fGetLine (LineNumber As Long)
      'This function copies a line of text specified by LineNumber from the edit control. The first
            line starts at zero.
    Const MAX_CHAR_PER_LINE = 80
```

```
    Const EM_GETLINE = &H400 + 20

    Text1.SetFocus
    Buffer$ = Space$(MAX_CHAR_PER_LINE)
    Pos% = SendMessage(GetFocus(), EM_GETLINE, LineNumber, Buffer$)
    aGetLine.Caption = "GetLine = " + Buffer$
    fGetLine = Pos%
End Function
```

```
Function fGetSel& ()
        'This function returns the starting/ending position of the current selected text.   This is the
                current location of the cursor if start is equal to ending. LOWORD-start position of
                selected text HIWORD-first no selected text
    Const EM_GETSEL = &H400 + 0

    Text1.SetFocus
    location& = SendMessage(GetFocus(), EM_GETSEL, 0&, 0&)
    ending% = location& \ 2 ^ 16
    starting% = location& Xor high%
    aGetSel.Caption = "Caret Location = " + Str$(starting%)
    fGetSel = location&
End Function
```

```
Function fLineFromChar& (CharPos&)
        'This function will return the line number of the line that contains the character whose
                location(index) specified in the third argument of SendMessage(). If the third
                argument is -1, then the number of the line that contains the first character of the
                selected text is returned. If start = end from GetSel, then the current caret location is
                used. Line numbers start at zero.
    Const EM_LINEFROMCHAR = &H400 + 25

    Text1.SetFocus
    Pos% = SendMessage(GetFocus(), EM_LINEFROMCHAR, CharPos&, 0&)
    aLineFromChar.Caption = "Current Line = " + Str$(Pos%)
    fLineFromChar = Pos%
End Function
```

```
Function fLineIndex (LineNumber As Long)
        'This function will return the number of bytes that precede the given line. The returned
                number reflects the CR/LF after each line. The third argument to SendMessage()
                specifies the line number, where the first line number is zero. If the third argument to
                SendMessage() is -1, then the current line number is used.
    Const EM_LINEINDEX = &H400 + 11

    Text1.SetFocus
    Pos% = SendMessage(GetFocus(), EM_LINEINDEX, LineNumber, 0&)
    aLineIndex.Caption = "#Char's before line = " + Str$(Pos%)
    fLineIndex = Pos%
End Function
```

```
Function fLineLength& (CharPos As Long)
        'This function will return the length of a line in the edit control. CharPos specifies the index of
                the character that is part of the line that you would like to find the length. If this
                argument is -1, the current selected character is used as the index.
    Const EM_LINELENGTH = &H400 + 17
```

```
    Text1.SetFocus
    Pos% = SendMessage(GetFocus(), EM_LINELENGTH, CharPos, 0&)
    aLineLength.Caption = "LineLength = " + Str$(Pos%)
    fLineLength = Pos%
End Function
```

```
Function fSetSel& (Pos%)
    'This function selects all characters in the current text that are within the starting and ending
        positions given by Location. The low word is the starting position and the high word
        is the ending position. If you set start to end, this can be used to position the cursor
        within the edit control.
    Const EM_SETSEL = &H400 + 1
    location& = Pos% * 2 ^ 16 + Pos%
    Text1.SetFocus
    X% = SendMessage(GetFocus(), EM_SETSEL, 0&, location&)
    fSetSel = Pos%
End Function
```

```
Function fReplaceSel (Buffer$)
    'This function will replace the current selected text with the new text specified in Buffer$. You
        must call SendMessage() with the EM_GETSEL constant to select text.
    Const EM_REPLACESEL = &H400 + 18
    Text1.SetFocus
    Pos% = SendMessage(GetFocus(), EM_REPLACESEL, 0&, Buffer$)
    aReplaceSel.Caption = "String inserted = " + Buffer$
    fReplaceSel = Pos%
End Function
```

References

Disk Copy of Code Examples from VB Programmer's Guide Manual

Article Number: Q76563

Summary:

The program examples from the "Microsoft Visual Basic: Programmer's Guide" version 1.0 manual are available in electronic form.

The files can be found in the Software/Data Library on CompuServe by searching for the filename VBEXAMPS, the Q number of this article, or S13182. VBEXAMPS was archived using the PKware file-compression utility. When you decompress VBEXAMPS, you will obtain the following files:

ADDMENU.FRM
COLOR.FRM
ERR.FRM
FILEINFO.FRM,
FILEINFO.TXT
INVEST.FRM
LAUNCH.FRM
NUMSYS.FRM,
RECEDIT.FRM
RECEDIT.TXT
TEXTEDIT.FRM
TEXTEDIT.TXT

More Information:

To use each sample program, do the following:

1. Start Visual Basic (VB.EXE).

2. From the File menu, choose Remove File, and remove the default Form1.

3. From the File menu, choose Add File.

4. Add the desired .FRM file. The file will appear in the project window.

5. If there is an associated .TXT file, load the contents of this file into the global module by first opening the Global module, then choosing Load Text from the Code menu.

6. Run the example.

The example programs are referenced in the "Microsoft Visual Basic: Programmer's Guide" version 1.0 manual on the following pages:

ADDMENU.FRM	Chapter 10, page 117
COLOR.FRM	Chapter 11, page 142
ERR.FRM	Chapter 19, page 280
FILEINFO.FRM, FILEINFO.TXT	Chapter 20, page 305
INVEST.FRM	Chapter 6, page 49

LAUNCH.FRM	Chapter 20, page 298
NUMSYS.FRM	Chapter 10, page 114
RECEDIT.FRM, RECEDIT.TXT	Chapter 21, page 335
TEXTEDIT.FRM TEXTEDIT.TXT	Chapter 21, page 319

Terminating Windows from a Visual Basic Application

Article Number: Q76981

Summary:

The Visual Basic **SendKeys** function cannot be used to close Program Manager in order to terminate Windows.

More Information:

You may want to terminate the current Windows session by closing the Program Manager from within a Visual Basic application. You may think that you can activate the Program Manager Control menu and send the appropriate key sequences using the Visual Basic **SendKeys** function. However, this method will not work because after the Close menu item is selected, a system modal dialog box is opened that prompts you to save changes to Program Manager. A system-modal dialog box locks out ALL other programs until it is satisfied. Thus, the keystroke you send via the **SendKeys** function will never reach that dialog box.

Steps to Reproduce Problem

1. Start a New Project in Visual Basic.

2. Draw a command button on the form.

3. In the command button Click event procedure, add the following code:

```
AppActivate("Program Manager")
SendKeys "%{ }{DOWN 5}{ENTER 2}", 0   'ALT, SPACE, DOWN 5, ENTER 2
```

4. Run the program.

Note that the Program Manager does not close. Choosing the OK button with the mouse will display a message stating "Can't quit at this time." Choosing the Cancel button will display the message "Cannot start more than one copy of the specified program." These messages are misleading, but are the result of attempting an unsupported action.

To correctly close Program Manager, you must use the **ExitWindows**() API function. You can declare this API function in the GLOBAL.BAS module.

For example:

1. Start a new project in Visual Basic.

2. Draw a command button on the form.

3. Add the following line to GLOBAL.BAS:

```
Declare Function ExitWindows% Lib "user" (ByVal dwReserved&, ByVal wReturnCode%)
```

4. Add the following line of code to the command button Click Procedure:

```
RetVal% = ExitWindows(0,0)
```

5. Run the program.

6. Click the command button.

The **ExitWindows**() API call initiates the standard Windows shutdown procedure. If all applications agree to terminate, the Windows session is terminated and control returns to MS-DOS. If the **ExitWindows**() API call fails (due to an open MS-DOS session or some other reason), FALSE is returned. You should check for this error and handle it appropriately.

VB Can Determine When a Shelled Process Has Terminated

Article Number: Q72880

Summary:

The **Shell** function initiates a process and returns back to the Visual Basic program. The process will continue indefinitely until you decide to stop it. Terminating the Visual Basic program will not cause the shelled process to terminate. However, you may not want this behavior if you want the Visual Basic program to wait until the shelled process has finished before continuing. This article describes a method by which a Visual Basic program will wait until a shelled process has terminated.

More Information:

By using the Windows API functions **GetActiveWindow**() and **IsWindow**(), your program can monitor the status of a shelled process. The API function **GetActiveWindow**() should be called immediately after the **Shell** function to get the window handle of the shelled process. This will work correctly only if you invoke the **Shell** function using a window style with focus, that is, window style 1, 2, or 3. By continually calling the API function **IsWindow**() from within a While loop, you can cause the Visual Basic program to wait until the shelled process has terminated. The Windows API function **IsWindow**() simply checks to make sure that the window associated with the handle found with **GetActiveWindow**() is still a valid window.

The Visual Basic program below uses the **Shell** function to execute the Windows Calculator accessory. The program below is an example of how to use the Windows API functions **GetActiveWindow**() and **IsWindow**() to wait until a shelled process has terminated before resuming execution.

Code Example

NOTE: The Declare statements for Windows API functions must be included in the Declarations section of the form or in the Global module. Assume that the following Declare statements are in the Declarations section of the default form (Form1).

```
Declare Function GetActiveWindow% Lib "User" ()
Declare Function IsWindow% Lib "User" (ByVal hWnd%)

    'Below is the code contained within the Form_Load event procedure of the default form
            (Form1):
```

```
Sub Form_Load ()
    x% = Shell("calc.exe", 1)
            'Get the window handle
        ShellWindowHWnd% = GetActiveWindow%()
            'Wait until the shelled process has been terminated.
        While IsWindow%(ShellWindowHWnd%)
            'process other Windows events.
            x% = DoEvents()
        Wend
    End
End Sub
```

How VB Can Determine If a Specific Windows Program Is Running

Article Number: Q72918

Summary:

To determine if a specific program is running, call the Windows API function **FindWindow**(). **FindWindow**() returns the handle of the window whose class is given by the lpClassname parameter and whose window name, or caption, is given by the lpCaption parameter. If the returned value is zero (0), the application is not running.

More Information:

By calling **FindWindow**() with a combination of a specific program's class name and/or the title bar caption, your program can determine whether that specific program is running.

When an application is started from the Program Manager, it registers the class name of the form. The window class provides information about the name, attributes, and resources required by your form. All Visual Basic forms have a class name of "ThunderForm". You can determine the class name of an application by using SPY.EXE, which comes with the Microsoft Windows 3.0 Software Development Kit (SDK).

If the window has a caption bar title, you can also use the title to locate the instance of the running application. This caption text is valid even when the application is minimized to an icon.

Because another instance of your Visual Basic program will have the same class name and may have the same title bar caption, you must use dynamic data exchange (DDE) to determine if another instance of your Visual Basic program is running. (This DDE technique is not shown in this article).

The following example shows three ways to determine if the Windows 3.0 Calculator is running. To create the program, do the following:

1. Create a form.

2. Declare the Windows 3.0 API function **FindWindow**() in the Global Declarations section of the Code window. The variables are declared as "Any" because you can pass either a pointer to a string, or a NULL value. You are responsible for passing the correct variable type. Note that the Declare statement should be entered on just one line.

```
Declare Function FindWindow% Lib "user" (ByVal lpClassName As Any, ByVal lpCaption As Any)
```

3. Add the following code to the form's Click event. This example demonstrates how you can find the instance of the application with a combination of the class name and/or the window's caption. In this example, the application will find an instance of the Windows 3.0 Calculator (CALC.EXE).

```
Sub Form_Click ()
    Const NULL = 0&

    lpClassName$ = "SciCalc"
    lpCaption$ = "Calculator"

    print "Handle = ";FindWindow(lpClassName$, NULL)
    Print "Handle = ";FindWindow(NULL, lpCaption$)
    Print "Handle = ";FindWindow(lpClassName$, lpCaption$)
End Sub
```

4. Run this program with CALC.EXE running and then without CALC.EXE running. If CALC.EXE is running, your application will print an arbitrary handle. If CALC.EXE is not running, your application will print the number zero as a handle.

Below are some class names of applications that are shipped with Windows 3.0:

Class Name	Application	
SciCalc	CALC.EXE	
CalWndMain	CALENDAR.EXE	
Cardfile	CARDFILE.EXE	
Clipboard	CLIPBOARD.EXE	
Clock	CLOCK.EXE	
CtlpanelClass	CONTROL.EXE	
XLMain	EXCEL.EXE	
Session	MS-DOS.EXE	
Notepad	NOTE.EXE	
PbParent	PBRUSH.EXE	
Pif	PIFEDIT.EXE	
PrintManager	PRINTMAN.EXE	
Progman	PROGMAN.EXE	(Windows Program Manager)
Recorder	RECORDER.EXE	
Reversi	REVERSI.EXE	
#32770	SETUP.EXE	
Solitaire	SOL.EXE	
Terminal	TERMINAL.EXE	
WFS_Frame	WINFILE.EXE	
MW_WINHELP	WINHELP.EXE	
#32770	WINVER.EXE	
MSWRITE_MENU	WRITE.EXE	

References

How to Access Windows Initialization Files Within Visual Basic

Article Number: 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 Visual Basic 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. Windows and Windows applications can use the information stored in these files to configure themselves to meet your needs and preferences. For a description of initialization files, read the WININI.TXT file that comes with Microsoft Windows 3.0.

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 keyname 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 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:

Parameter	Type/Description
pAppName$	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 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 3.0. 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 3.0) 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.

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 Determine Multiple Instances of a VB Application

Article Number: Q75641

Summary:

Using Windows version 3.0 API function calls, you can determine if another instance of your application is running. Using the same API calls, you can also determine how many instances of an application are running.

More Information:

You can use two Windows API function calls to determine how many instances of your application are running. This can be useful if you want to limit how many copies of your application can run at once.

The Windows KERNEL dynamic-link library (DLL) defines two functions called **GetModuleHandle**() and **GetModuleUsage**(). **GetModuleUsage**() uses the handle returned from **GetModuleHandle**() to determine how many instances of your application are running. Below is a definition of each function:

GetModuleHandle%(lpProgramName$)

Parameter	Type/Description
lpProgramName$	Points to a null-terminated character string that specifies the program.
Return Value	The return value identifies the program if the function is successful. Otherwise, the return value is zero.

GetModuleUsage%(hProgram%)

Parameter	Type/Description
hProgram%	Identifies the program or an instance of the program. This value can be determined with a call to **GetModuleHandle**().
Return Value	The return value specifies the reference count of the program.

Example

The following application is an example of how to limit an application to a single instance:

1. Create a form called Form1.

2. Within the Global Declaration section of the form, declare the following Windows API functions:

```
Declare Function GetModuleHandle% Lib "Kernel" (ByVal lpProgramName$)
Declare Function GetModuleUsage% Lib "Kernel" (ByVal hProgram%)
```

3. Within the Form_Load event add the following code:

```
Sub Form_Load ()
    hw% = GetModuleHandle("Project.EXE")
    If GetModuleUsage(hw%) > 1 Then
        MsgBox "This program is already loaded!", 16
        End
    End If
```

```
End Sub
```

4. Compile the program as PROJECT.EXE

5. Run PROJECT.EXE from the Program Manager.

6. Run a second instance of PROJECT.EXE. It should display a message box and terminate.

References

How to Print the ASCII Character Set in Visual Basic

Article Number: Q75857

Summary:

The default font used by Visual Basic is the standard ANSI character set. To display the ASCII character set, which is more commonly used by MS-DOS mode applications, you must call two different Windows API functions, **GetStockObject**() and **SelectObject**().

More Information:

Windows supports a second character set, referred to as the OEM character set. This is generally the character set used internally by MS-DOS for screen display at the MS-DOS prompt. The character codes 32 to 127 are normally identical for the OEM, ASCII, and ANSI character sets. The ANSI characters represented by the remaining character codes (0 to 31 and 128 to 255) are generally different from characters represented by the OEM and ASCII character sets. The OEM and ASCII character sets are identical for these ranges. Under the ASCII and OEM character sets, the character codes 128 to 255 correspond to the extended ASCII character set, which includes line drawing characters, "graphics" characters, and special symbols. The characters represented by this range of character codes generally differ between the ASCII (or OEM) and ANSI character sets.

To change the selected font from ANSI to the OEM ASCII font, you must get a handle to the OEM character set by calling **GetStockObject**(). When this handle is passed as an argument to **SelectObject**(), the ANSI font will be replaced by the OEM ASCII font. This API function also returns the handle to the font object previously used. Once you are through displaying the desired characters, you should call **SelectObject**() again to reselect the original font object.

> Note: There is also an API function called **DeleteObject**(). This function need not be called to delete a stock object. The purpose of this API function is to delete objects loaded with the API function **GetObject**().

The three functions are described as follows:

GetStockObject% (nIndex%)

Parameter	Type/Description
nIndex%	Specifies the type of stock object desired. Use the constant OEM_FIXED_FONT to retrieve the handle to the OEM character set. The value of this constant is 10.
Return Value	The return value identifies the desired logical object if the function is successful. Otherwise, it is NULL.

SelectObject% (hDC%, hObject%)

Parameter	Type/Description
hDC%	Identifies the device context.
hObject%	Identifies the object to be selected. Use the return value from **GetStockObject**% (above) to select the OEM character set.
Return Value	The return value identifies the handle to the object previously used. This value should be saved in a variable such that **SelectObject**() can be called again to restore the original object used. It is NULL if there is an error.

The following steps describe how to create a program example that demonstrates how to print ASCII characters.

1. Run Visual Basic.

2. From the File menu, choose New project (ALT, F, N).

3. Create a command button called Command1 on the default form (Form1).

4. Add the following declarations to the General Declarations section of Form1.

```
Declare Function GetStockObject% Lib "GDI" (ByVal nIndex%)
Declare Function SelectObject% Lib "GDI" (ByVal hdc%, ByVal hObject%)
```

5. Place the following code in the Command1 click event procedure:

```
Sub Command1_Click ()
    Const OEM_FIXED_FONT = 10
    Const PIXEL = 3
      'handle to the OEM font object
    Dim hOEM As Integer
    Dim Y, H As Single
      'save the scale mode so that you can reset later
    Saved% = Form1.ScaleMode
    Form1.ScaleMode = PIXEL
      'Get the character height and subtract the external leading
    H = Form1.TextHeight(Chr$(200)) - 1
      'get the handle to the desired font
    hOEM = GetStockObject(OEM_FIXED_FONT)
      'select the object relating to the font handle, if successful then print the desired characters.
    PreviousObject% = SelectObject(Form1.hDC, hOEM)
    If PreviousObject% Then
        Form1.CurrentX = 10: Form1.CurrentY = 10
        Print Chr$(201); Chr$(187);
        Form1.CurrentX = 10:
        Form1.CurrentY = Form1.CurrentY + H
        Print Chr$(200); Chr$(188)
      'Reinstate previous font
        hOEM = SelectObject(Form1.hDC, PreviousObject%)
    Else
      'SelectObject was unsuccessful
        MsgBox "Couldn't find OEM fonts", 48
    End If

      'reset the scale mode
    Form1.ScaleMode = Saved%
End Sub
```

6. From the Run menu, choose Start.

7. Click the Command1 button.

When the Command1 button is clicked or selected, a small box with a double border will be drawn in the upper-left corner of the screen. The box is drawn using characters associated with the extended ASCII character set.

References

How to Trap VB Form Lost Focus with GetActiveWindow API

Article Number: Q69792

Summary:

The LostFocus event in Microsoft Visual Basic is useful when transferring control within an application. However, no global routine exists to check for the entire form losing the focus. One method to use to check whether your Visual Basic application has lost the focus is by periodically checking the Windows API function **GetActiveWindow**() in a Visual Basic timer event, as explained below.

More Information:

The only way that Visual Basic provides a check for loss of focus on a form or control is by triggering the LostFocus event. A form does support a LostFocus event; however, a form will get the focus only if there are no controls on that form. The focus goes to the controls on a form, and when you click any other visible form, the control's LostFocus procedure will be called. A control's LostFocus procedure will also be called when another control on the form is activated. To perform a routine that occurs only when the form loses the focus requires careful management of what generated a LostFocus event on each control (such as setting a flag if another control's Click event was called).

For a simpler method to check if a whole form has lost the focus, you can call the Windows API function **GetActiveWindow**(), located in USER.DLL in Windows 3.0. The **GetActiveWindow**() API call returns the window handle of the currently active window, which is the new window that you last clicked anywhere in Microsoft Windows. In a timer event procedure for the form, call **GetActiveWindow**() and compare the handle of the currently active Window with the handle of the form window (Form1.hWND). If the handle differs, you know the form has lost the focus. The following program example demonstrates this technique.

Program Example

This single-form example will print "Lost Focus" on the form when you click a different window (such as when you click another program running in Windows).

1. In Visual Basic, draw one timer control (Timer1) and one command button (Command1) on a single form (Form1).

2. From the VB.EXE Code menu, choose View Code, and enter the following code for Form1, using (general) from the Object box, and (declarations) from the procedure box:

```
Declare Function GetActiveWindow% Lib "User" ()
Dim FOCUS As Integer
Const TRUE = -1
Const FALSE = 0
```

3. From the Object box, choose Timer1, and from the procedure box, choose Timer, and then put the following code in the Timer1_Timer procedure:

```
Sub Timer1_Timer ()
    If FOCUS = TRUE Then
      'Compare the handle to the handle of current active Window with of the Form1 window:
        If GetActiveWindow() <> Form1.hWND Then
      'Do form's lost-focus routines here.
            Print "Lost Focus"
            FOCUS = FALSE
```

```
            End If
        End If
    End Sub
```

4. You must set FOCUS = TRUE in the Click event procedure of every control on the form, as follows:

5. From the Object box, choose Command1, and from the procedure box, choose Click, then put the following code in the Command1_Click procedure:

```
Sub Command1_Click ()
    FOCUS = TRUE
End Sub
```

6. Double-click Form1 (at design time) and enter the following code for the Form_Click procedure:

```
Sub Form_Click ()
    FOCUS = TRUE
    Timer1.Interval = 10
End Sub
```

You can now run the program.

References

Program Example for Communication Port Support in Visual Basic

Article Number: Q75856

Summary:

A sample program is available to show how a Visual Basic program can use Windows API functions for serial port communications. It can be downloaded from CompuServe or obtained by calling Microsoft Product Support Services.

On CompuServe, VBCOMDEM can be found in the Software/Data Library by searching for the word VBCOMDEM, the Q number of this article, or S13150. VBCOMDEM was archived using the PKware file-compression utility. When you decompress VBCOMDEM, you will obtain the following 18 files:

ABOUTDLG.FRM, ABOUTDLG.TXT, CHKLIST.CPS, COMMDEMO.EXE, COMMDEMO.FRM, COMMDEMO.MAK, COMMDEMO.TXT, EVENTDLG.FRM, EVENTDLG.TXT, GLOBAL.BAS, LINEDLG.FRM, LINEDLG.TXT, MODULE1.BAS, MODULE1.TXT, PORTDLG.FRM, PORTDLG.TXT, RECEIVE.FRM, RECEIVE.TXT

More Information:

In the Visual Basic environment (VB.EXE), you can load the files in this sample program by choosing Open Project from the File menu, and selecting the COMMDEMO.MAK file.

You can also run COMMDEMO.EXE in Windows as a separate program that requires the Visual Basic run-time file VBRUN100.DLL.

This sample program is only a starting point, and does not utilize all of the serial communications API functions available through Windows. This simple example uses Windows API "comm" functions, such as **OpenComm**(), **CloseComm**(), **ReadComm**(), and **WriteComm**(). You are free to modify and extend the program to suit your specific needs. The COMMDEMO program has no error trapping and makes no allowances for noisy communication lines or handshaking errors. Should an error occur, Windows will suspend all reading from the communications port until you clear the error by calling the Windows API function **GetCommError**().

To modify or understand this program example, you must have a reference manual for the Windows API routines.

References

How to Create and Use a Custom Cursor in Visual Basic; Win SDK

Article Number: 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 Visual Basic 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 application. Below are the steps necessary to a create a custom cursor and a Visual Basic application to use the custom cursor.

More Information:

Setting a custom cursor in a Visual Basic 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. Farther below are the steps necessary to create a Visual Basic program to use the cursor resources. If you do not have the Windows SDK but have a precompiled 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 WinSDK Paint, 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 application that uses the cursor resources created in the steps above.

Warning : When running the Visual Basic program created by following the steps below, it is important to terminate the application from the Control menu, NOT by using the Run End option on 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. Also, 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.

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

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
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 APISetFocus 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 = GetFocus()
      'Get the Window handle of Command1.
    Command1.SetFocus
    Command1hWnd = GetFocus()
      'Restore the focus to the control with the input focus
    r = APISetFocus(CurrHwnd)
      'One millisecond
    Timer1.Interval = 1
    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:
            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.

References

How to Create Pop-Up Menus on a Visual Basic Form

Article Number: Q71279

Summary:

Visual Basic can call the Windows API function **TrackPopupMenu**() to display a specified menu at the location on the screen that the user clicks with the mouse.

More Information:

The **TrackPopupMenu**() function displays a "floating" pop-up menu at the specified location and tracks the selection of items on the pop-up menu. A floating pop-up menu can appear anywhere on the screen. The hMenu parameter specifies the handle of the menu to be displayed; the application obtains this handle by calling **GetSubMenu**() to retrieve the handle of a pop-up menu associated with an existing menu item.

TrackPopupMenu() is defined as follows:

TrackPopupMenu (hMenu%, wFlags%, x%, y%, rRes%, hWnd%, lpRes&)

Parameter	Type/Description
hMenu%	Identifies the pop-up menu to be displayed.
wFlags%	Not used. This parameter must be set to zero.
x%	Specifies the horizontal position in screen coordinates of the left side of the menu on the screen.
y%	Specifies the vertical position in screen coordinates of the top of the menu on the screen.
nRes%	Is reserved and must be set to zero.
hWnd%	Identifies the window that owns the pop-up menu.
lpRes&	Is reserved and must be set to NULL.

The supporting Windows API functions needed to support the arguments to **TrackPopupMenu**() are:

GetMenu(hWnd%)

Parameter	Type/Description
hWnd%	Identifies the window whose menu is to be examined.
Return Value	NULL if the given window has no menu. The return value is undefined if the window is a child window.

GetSubMenu(hMenu%, nPos%)

Parameter	Type/Description
hMenu%	Identifies the menu.
nPos%	Specifies the position in the given menu of the pop-up menu. Position values start at zero for the first menu item.

GetSubMenu() returns a value that identifies the given pop-up menu. The return value is NULL if no pop-up menu exists at the given position.

To create a pop-up menu within Visual Basic, define a menu system with the Menu Design window. The following is an example of a menu system:

Caption	CtlName	Indented
File	M_File	No
New	M_New	Once
Open	M_Open	Once
Close	M_Close	Once
Exit	M_Exit	Once
Help	M_Help	No

Within the General Declaration section of your Code window, declare the following:

```
Declare Function TrackPopupMenu% Lib "user"(ByVal hMenu%, ByVal wFlags%, ByVal X%,
        ByVal Y%, ByVal r2%, ByVal hWnd%, ByVal r1&)

Declare Function GetMenu% Lib "user" (ByVal hwnd%)

Declare Function GetSubMenu% Lib "user" (ByVal hMenu%, ByVal nPos%)
```

Note: Each Declare statement above must be located on just one line.

Place the following code in the form's MouseUp event procedure:

```
Sub Form1_MouseUp (Button As Integer, Shift As Integer, X As Single, Y As Single)
      'The above Sub statement must be concatenated onto one line.
    Const PIXEL = 3
    Const TWIP = 1
    ScaleMode = PIXEL
    InPixels = ScaleWidth
    ScaleMode = TWIP
    IX = (X + Left) \ (ScaleWidth \ InPixels)
    IY = (Y + (Top + (Height - ScaleHeight -(Width - ScaleWidth)))) \ (ScaleWidth \ InPixels)
      'The above IY = ... statement must be concatenated onto one line.
    hMenu% = GetMenu(hwnd)
    hSubMenu% = GetSubMenu(hMenu%, Button - 1)
    R = TrackPopupMenu(hSubMenu%, 0, IX, IY, 0, hwnd, 0)
End Sub
```

When you run the program, clicking anywhere on Form1 will display the first menu on your menu bar at that location.

References

How to Emulate QuickBasic's SOUND Statement in Visual Basic

Article Number: Q71102

Summary:

The SOUND statement found in Microsoft QuickBasic is not implemented within Microsoft Visual Basic. You can perform sound through a Windows 3.00 API call that is equivalent to the QuickBasic SOUND statement.

More Information:

The QuickBasic version of the SOUND statement can be executed by calling several Windows 3.00 API function calls. Within Windows, you must open up a **VoiceQueue**() with the **OpenSound**() call routine. Using the function **SetVoiceSound**(), place all of the values corresponding to the desired frequencies and durations. Once the **VoiceQueue**() has the desired frequencies and durations, you start the process by calling **StartSound**(). After the sounds have been played, you must free up the **VoiceQueue**() by calling **CloseSound**(). If you plan on placing a large amount of information into the **VoiceQueue**(), you may need to resize the **VoiceQueue**() buffer by calling the **SetVoiceQueueSize**() function.

After executing the **StartSound**() function, you cannot place any more sound into the **VoiceQueue**() until the **VoiceQueue**() is depleted. Placing more sound into the queue will overwrite any information that was previously in the **VoiceQueue**(). If you are going to place sound into the **VoiceQueue**() after a **StartSound**() statement, you will need to call **WaitSoundState**() with an argument of one. When **WaitSoundState**() returns NULL, the **VoiceQueue**() function is empty and processing can continue.

Below is an example of using the Windows API function calls, which will imitate the QuickBasic SOUND statement:

In the General section, place the following:

```
Declare Function OpenSound% Lib "sound.drv" ()
Declare Function VoiceQueueSize% Lib "sound.drv"(
        ByVal nVoice%,
        ByVal nBytes%)
Declare Function SetVoiceSound% Lib "sound.drv"(
        ByVal nSource%,
        ByVal Freq&,
        ByVal nDuration%)
Declare Function StartSound% Lib "sound.drv" ()
Declare Function CloseSound% Lib "sound.drv" ()
Declare Function WaitSoundState% Lib "sound.drv" (
        ByVal State%)
```

Note: All Declare statements above each must be placed on one line.

The **SetVoiceSound**() function takes two arguments. The first variable, Freq, is a two word parameter. The high word will hold the actual frequency in hertz. The low word will hold the fractional frequency. The formula, X * 2 ^ 16, will shift the variable "X" into the high word location. The second variable, Duration%, is the duration in clock ticks. There are 18.2 tick clicks per second on all Intel computers.

The following simple example shows how you can place several frequencies and durations into the **VoiceQueue**() function before starting the sound by calling the **StartSound**() function:

```
Sub Form_Click ()
    Suc% = OpenSound()
      'Frequency = 100hz
    S% = SetVoiceSound(1, 100 * 2 ^ 16, 100)
      'Frequency = 90 hz
    S% = SetVoiceSound(1, 90 * 2 ^ 16, 90)
      'Frequency = 80 hz
    S% = SetVoiceSound(1, 80 * 2 ^ 16, 90)
    S% = StartSound()
    Succ% = CloseSound()
End Sub
```

The following is another simple example, which creates a siren sound:

1. Within the general section, place the following Sound procedure:

```
Sub Sound (ByVal Freq as Long, ByVal Duration%)
      'Shift frequency to high byte.
    Freq = Freq * 2 ^ 16
    S% = SetVoiceSound(1, Freq, Duration%)
    S% = StartSound()
    While (WaitSoundState(1) <> 0): Wend
End Sub
```

2. Place the code below into any event procedure. The example below uses the Form_Click event procedure. Clicking any position on the form will create a police siren.

```
Sub Form_Click ()
    Suc% = OpenSound()
    For j& = 440 To 1000 Step 5
        Call Sound(j&, j& / 100)
    Next j&
    For j& = 1000 To 440 Step -5
        Call Sound(j&, j& / 100)
    Next j&
    Succ% = CloseSound()
End Sub
```

References

Simulating ON KEY Key Trapping with KeyDown Event in VB

Article Number: Q75858

Summary:

Although there is no ON KEY GOSUB statement in Visual Basic, the effect of ON KEY event handling can be achieved in Visual Basic. Visual Basic forms and controls that are able to get focus within Visual Basic have a [form/control]_KeyDown event procedure that can simulate the effects of the ON KEY statements of MS-DOS-based Basics. The process of using the KeyDown event procedure is more powerful and more flexible than the ON KEY statement.

More Information:

If you press a key while one of your Visual Basic forms or controls has the focus, the KeyDown event procedure for that form or control will be executed. Within the KeyDown event procedure, you can call a global procedure and pass the actual key states to the global procedure to create the same effect as the ON KEY event trapping that is performed in the MS-DOS-based Basic.

You can also pass the control or form where the KeyDown event occurred to the global procedure. Passing the control or form itself will allow the global procedure to tell what control or form called the global procedure.

To create a small example, do the following:

1. Within Visual Basic's Project window, double-click the GLOBAL.BAS file to bring up the Code window. From Visual Basic's Code menu, choose Load Text. Load the CONSTANTS.TXT file that came with Visual Basic. Note: If you already have text within the GLOBAL.BAS file, you will need to create another module, add the CONSTANTS.TXT to this file, and then cut and paste to the GLOBAL.BAS file.

2. Create two text boxes on a form.

3. In the Text1_KeyDown event procedure, add the following code:

```
Call OnKeyGoSub(KeyCode, Shift, Text1)
```

4. In the Text2_KeyDown event procedure, add the following code:

```
Call OnKeyGoSub(KeyCode, Shift, Text2)
```

5. Add a Label to Form1 called Label1.

6. At the global Declaration section for form1, add the following procedure:

```
Sub OnKeyGoSub (KeyCode%, Shift%, Ctrl As Control)
    Select Case KeyCode%
        Case KEY_MENU: Key$ = ""
        Case KEY_SHIFT: Key$ = ""
        Case KEY_CONTROL: Key$ = ""
        Case KEY_F1: Key$ = " F1 "
        Case KEY_UP: Key$ = " UP key"
        Case KEY_CAPITAL: Key$ = "CAP LOCKS"
        Case Else: Key$ = Chr$(KeyCode%)
    End Select
    Select Case Shift%
```

```
            Case SHIFT_MASK: Shft$ = "Shift"
            Case ALT_MASK: Shft$ = "Alt"
            Case CTRL_MASK: Shft$ = "Ctrl"
            Case Else: Shft$ = ""
        End Select
        ControlName$ = Ctrl.CtrlName
        Label1.Caption = "Key:" + Shft$+ " "+ Key$ +" from "+ ControlName$
    End Sub
```

7. Run the program. Move back and forth between the two text boxes using either the TAB key or the mouse. Experiment with any key in combination with the ALT, CTRL, and SHIFT keys. Also try the F1 key and the UP ARROW key.

This example is limited but shows how you could simulate the ON KEY statements or key trapping within Visual Basic. Placing the call to the key trap procedure within any KeyDown event procedure will simulate the ON KEY statement.

How to Set Hourglass Mouse Pointer in VB Program During Delays

Article Number: Q71105

Summary:

During operations that take time, you may decide to display the hourglass mouse cursor. This will let the user know that the computer is performing an operation that may take some noticeable time, and that no user input will be immediately processed during this time.

More Information:

The MousePointer property can be set during design time or during run time. During operations that may take some noticeable amount of time, the MousePointer should be set to the hourglass pointer. This can be performed using the Screen.MousePointer property. Before setting the mouse pointer to the hourglass, you should save the present mouse pointer so that when you are through with the hourglass cursor, you can reinitialize it to the previous pointer. Below is an example of setting the pointer to the hourglass cursor:

```
Sub Form_Click ()
        'Save mouse pointer.
    SavedPointer = Screen.MousePointer
      '11# = hourglass.
    Screen.MousePointer = 11
      'Some lengthy operation.
    For i = 1 To 10000: Next i
      'set to previous mouse pointer.
    Screen.MousePointer = SavedPointer
End Sub
```

Determining Whether TAB or Mouse Gave a VB Control the Focus

Article Number: Q75411

Summary:

You can determine whether a Visual Basic control received the focus from a mouse click or a TAB keystroke by calling the Windows API function **GetKeyState**() in the control's GotFocus event procedure. By using **GetKeyState**() to check if the TAB key is down, you can determine if the user tabbed to the control; if the TAB key was not used (and the control does not have an access key), the user must have clicked the control with the mouse to set the focus.

More Information:

The **GetKeyState**() Windows API function takes an integer parameter containing the virtual key code for the desired key states. **GetKeyState**() returns an integer. If the return value is negative, the key has been pressed.

The following is a code example. To use this example, start with a new project in Visual Basic. Add a text box and a command button to Form1. Enter the following code in the project's GLOBAL.BAS module:

```
        'Global Module
Declare Function GetKeyState% Lib "User" (ByVal nVirtKey%)
Global Const VK_TAB = 9
```

Add the following code to the GotFocus event procedure for the Text1 text box control:

```
Sub Text1_GotFocus()
    If GetKeyState(VK_TAB) < 0 Then
        Text1.SelStart = 0
        Text1.SelLength = Len(Text1.Text)
        Else
        Text1.SelLength = 0
    End If
End Sub
```

Run the program. If you use the TAB key to move the focus from the command button to the text box, you should see the text in the text box selected. If you change the focus to the text box by clicking it with the mouse, the text will not be selected.

If the control has an access key (assigned by using an ampersand [&] in the control's caption property), then you may also want to check the state of the virtual ALT key using **GetKeyState**() to see if the user changed the focus using the access key. The virtual key code for ALT, actually known as VK_MENU, is 12.

Sending Keystrokes from Visual Basic to an MS-DOS Application

Article Number: Q77394
MODIFIED: 20-DEC-1991

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. One method to work around this limitation is presented below.

More Information:

The Visual Basic **SendKeys** function can send keystrokes to the currently active window (as if the keystrokes had been typed at the keyboard). The "Microsoft Visual Basic: Language Reference" version 1.0 manual states that you are not allowed to send keystrokes to a non-Windows application. This is correct, but 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 an MS-DOS session (running in a window).

2. Start Visual Basic.

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 CtlName 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
        'Append a <CR>
Clipboard.SetText DosKeys.Text + Chr$(13)
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 immediately following the SendKeys will interrupt the processing. This AppActivate 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 to the DosKeys text box (for example, "DIR").

11. Click 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 program, you should change the progname assignment from "Microsoft Visual Basic" to the executable filename. Also, if you would like to see the text being placed onto the Clipboard, you can open the Windows Clipboard viewer.

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

Article Number: Q78001
MODIFIED: 19-NOV-1991

Summary:

By calling the Windows API functions **GetWindow**(), **GetWindowText**(), and **GetWindowTextLength**(), you can get the window titles of all windows (visible and invisible) loaded under Windows. The list of all of the window titles under Windows is known as the master list. The Windows Task Manager contains a list of the window titles for each of the top-level windows (normally one per application). This list is know as the Task List.

Farther below is a sample program that demonstrates how to AppActivate an application that is available from a list of top-level windows.

More Information:

The Task List is generally a subset of the master list. The Windows API functions only support methods of getting the master list, not the Task List. However, from the master list you can get a list of all top-level windows that closely resembles the Task List. The only difference is that the list containing the top-level windows may have more entries than the Task List. This is because an application can remove itself from the Task List; however, it will be included as part of the master list.

The example below demonstrates how to get the names of all top-level windows. The names of child windows can also be obtained by calling the **GetWindow**() API function with the GW_CHILD constant. Although the code example only provides an example of using the constants GW_HWNDFIRST and GW_HWNDNEXT as arguments to **GetWindow**(), the value of the other constants, such as GW_CHILD, are provided in the code.

To construct a sample program that demonstrates how to load the Task List into a Visual Basic combo box:

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

2. Change the caption property of Form1 to AppActivate.

3. Add the following controls to Form1 and change the CtlNames as indicated in the chart below:

Control	Name	CtlName
Label Control	Label1	Label1
Combo Box	Combo1	Combo_ListItem
Command Button	Command1	Command_Ok

4. Change the Caption properties of the controls as follows:

Control	Name	CtlName
Label Control	Label1	Application to AppActivate:
Command Button	Command_OK	OK

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

```
DefInt A-Z
    ' Windows API function declarations
```

```
Declare Function GetWindow Lib "user" (
            ByVal hWnd,
            ByVal wCmd) As Integer
Declare Function GetWindowText Lib "user" (
            ByVal hWnd,
            ByVal lpSting$,
            ByVal nMaxCount) As Integer
Declare Function GetWindowTextLength Lib "user" (
            ByVal hWnd) As Integer

Const False = 0
Const True = Not False

        ' Declare constants used by GetWindow
Const GW_CHILD = 5
Const GW_HWNDFIRST = 0
Const GW_HWNDLAST = 1
Const GW_HWNDNEXT = 2
Const GW_HWNDPREV = 3
Const GW_OWNER = 4
```

6. Add the following code in the Form_Load event procedure of Form1:

```
Sub Form_Load ()
    Call LoadTaskList
      ' Check to see if any items are in the task list, if not end the program.
    If Combo_ListItem.ListCount > 0 Then
        Combo_ListItem.Text = Combo_ListItem.List(0)
    Else
        MsgBox "Nothing found in task list", 16, "AppActivate"
        Unload Form1
    End If
End Sub
```

7. Enter the following code in the Click event procedure of the Command_Ok button:

```
Sub Command_Ok_Click ()
        ' Get the item selected from the text portion of the combo box
    f$ = Combo_ListItem.Text
      ' Simply resume if an "Illegal function call" occurs on the AppActivate statement
    On Local Error Resume Next
    AppActivate f$
End Sub
```

8. Enter the following code under the general declarations section of Form1:

```
Sub LoadTaskList ()
        'Get the hWnd of the first item in the master list so we can process the task list entries (top-
                level only)
    CurrWnd = GetWindow(Form1.hWnd, GW_HWNDFIRST)
      'Loop while the hWnd returned by GetWindow is valid
    While CurrWnd <> 0
      ' Get the length of the task name identified by CurrWnd in the list
        Length = GetWindowTextLength(CurrWnd)
      ' Get the task name of the task in the master list
```

```
            ListItem$ = Space$(Length + 1)
            Length = GetWindowText(CurrWnd, ListItem$, Length + 1)
        ' If there is an actual task name in the list, add the item to the list
            If Length > 0 Then
                Combo_ListItem.AddItem ListItem$
            End If
        ' Get the next task list item in the master list
            CurrWnd = GetWindow(CurrWnd, GW_HWNDNEXT)
        ' Process Windows events
            x = DoEvents()
    Wend
End Sub
```

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

From the combo box, select the window title of an application currently running under Windows. Choose the OK button to activate the application.

References

Visual Basic Code Window Hides Split View If Resized

Article Number: Q79057
Modified: 20-DEC-1991

Summary:

The Visual Basic development environment (VB.EXE) behaves unexpectedly when a split Code window is resized. Instead of proportionally resizing the two subwindows along with the parent window, the lower split view is obscured. The only indication that a split window is in effect is that the horizontal scroll bar and the bottom of the vertical scroll bar are also obscured.

More Information:

To work around this problem, resize the Code window to a convenient size before splitting it.

Steps to Reproduce Problem

1. Open a Code window.

2. Create a split Code window by placing the cursor between the Code window header and the top of the Code window and dragging downward.

3. Resize the Code window to a smaller size (whether from the top down or from the bottom up).

The result is that the lower window is hidden, including any breakpoints you were trying to track (for example, while watching a breakpoint set in each window).

In VB, Format$ Using # for Digit Affects Right Justification

Article Number: Q79094
Modified: 20-DEC-1991

Summary

The number (or pound) sign (#) does not serve as a placeholder for blank spaces when it is used with the Format$ function to reformat numbers as strings. If a number sign placeholder is not filled by a digit, Format$ truncates that digit position and will not replace that position with a space. This may be undesirable behavior if you are attempting to right justify the numeric digits within the string. This behavior is by design.

The number sign (#) placeholder is handled differently between the Visual Basic Format$ function and the Print Using statement found in other Basic products. In the case of the Print Using statement, a number sign placeholder is replaced by a space when no numeric digit occupies that position. By using the Print Using statement, you can right justify a formatted numeric string using the number sign as placeholders for the number. Note that Visual Basic does not support the Print Using statement. Additional code is needed to right justify a string using the Format$ function. An example is given farther below.

More Information:

Page 121 of the "Microsoft Visual Basic: Language Reference" for version 1.0 regarding the Format$ function is unclear on how the number sign is handled. When there is no numeric digit to fill the number sign placeholder, the manual does not clarify whether the number sign is replaced by a space or truncated. The documentation should be changed to reflect how the number sign is handled by the Format$ function.

The Print Using statement supported in other Basic products allows the use of the number sign as a placeholder for leading or trailing spaces, as follows:

```
Print Using "##0.00"; myvar
```

The above example will cause two leading spaces to be added to the resulting string representation of the variable "myvar" when the value of myvar is printed to the screen.

When used with the Visual Basic Format$ function, the same number sign format switch (#) does not work as a placeholder for spaces:

```
mystring$ = Format$(myvar , " ##.## ")
```

The Visual Basic Format$ function yields a formatted string representation of myvar with no leading spaces. This may not be the result you expected (for example, when myvar = 1.23). You may have expected the formatted result to have one leading space, allowing you to right justify the number, but no leading space is added.

The following code sample will produce an output of right-aligned numbers in QuickBasic version 4.5.

```
a = 1.23
b = 44.56
Print Using "##.##"; a
Print Using "##.##"; b
```

The following code sample will produce an output of left-aligned numbers in Visual Basic:

```
Sub Form_Click ()
    a = 1.23
    b = 44.56
    num1$ = Format$(a, "##.##")
    num2$ = Format$(b, "##.##")
    Print num1$
    Print num2$
End Sub
```

Click on the form to print the numbers. These numbers will be left aligned, instead of right aligned, as may be desired.

A workaround is to use a monospace font, such as Courier, and use the Len function to determine how many spaces need to be added to the left of the string representation of the number in order to right align the result.

```
Sub Form_Click ()
      'longest number expected
    desired = 5
    a = 1.23
    b = 44.56
       'Select a fixed-spaced font
    FontName = "Courier"
      'This converts number to a string
    num1$ = Format$(a, "#0.00")
       '2 decimal places and a leading 0
    num2$ = Format$(b, "#0.00")
    If (desired - Len(num1$)) > 0 Then
        num1$ = Space$(desired - Len(num1$)) + num1$
    End If
    If (desired - Len(num2$)) > 0 Then
        num2$ = Space$(desired - Len(num2$)) + num1$
    End If
    Print num1$
    Print num2$
End Sub
```

VB "Illegal Function Call" Under Windows Low Memory Condition

Article Number: Q79097
Modified: 20-DEC-1991

Summary:

Under extremely low memory conditions in Windows, Visual Basic may generate an "Illegal function call" error message even though there is no error in the code.

More Information:

To avoid the "Illegal function call" error, free some memory by closing other Windows applications.

Steps to Reproduce Problem

When Windows system memory resources are nearly exhausted, you can perform the following steps to reproduce the error:

1. Start Visual Basic.

2. From the Run menu, choose Start (ALT, R, S).

3. From the Run menu, choose Break (ALT, R, K).

4. From the Window menu, choose Immediate Window (ALT, W, I).

5. Enter the following code in the Immediate window:

```
MsgBox "hi"
```

If memory is sufficiently low, an "Illegal function call" error message will be displayed.

VB Multiline Text Box Memory Not Freed When Form Is Unloaded

Article Number: Q79115
Modified: 20-DEC-1991

Summary:

Repeatedly loading and unloading a multiline text box in a Microsoft Visual Basic program will result in a loss of memory proportional to the amount of text in the text box for each unload. This memory is not recovered until you terminate the Visual Basic environment (if running under the VB.EXE environment), or end your compiled program (if running a compiled .EXE). With large amounts of text or many loads and unloads, an "Out of Memory" error can occur.

This problem occurs only when the form is loaded and then unloaded. Hiding a form containing a multiline text box will not result in a loss of memory. This problem does not occur with a single line text box.

More Information:

Steps to Reproduce Problem

1. Start Visual Basic, or from the File menu, choose New Project.

2. From the File menu, choose New Form to add a second form to the project.

3. Create a text box named "Text1" on Form2. Set the Text1 MultiLine property to True.

4. Add the following code to Form1:

```
Sub Form_Click
    Form2.Show
End Sub
```

5. Add the following code to Form2:

```
Sub Form_Load
    Open "<any text file>" For Input As #1
    While Not Eof(1)
        Input #1, a$
        b$ = b$ + Chr$(13) + Chr$(10) + a$
    Wend
    Close
    Text1.Text = b$
End Sub
```

```
Sub Form_Click
    Unload Form2
End Sub
```

6. Run the program. In the Windows Program Manager, choose About Program Manager from the Help menu to check the amount of free memory.

7. Click Form1; Form2 will load and display the text from the text file.

8. Click Form2 to unload Form2; focus should return to Form1. Go to Program Manager and check free memory again -- the result is a loss of memory proportional to the amount of text in the

multiline text box.

Showing and unloading Form2 will continue to use memory each time it is performed. The lost memory is not recovered until you completely exit VB.EXE or terminate your compiled Visual Basic application. This problem does not occur when using a single line text box.

VB Extra Resize Event When WindowState Not Normal (Not 0)

Article Number: Q79116
Modified: 20-DEC-1991

Summary:

An extra Resize event will occur when a Visual Basic form is unloaded if that form's WindowState property is set to either 1 (Minimized) or 2 (Maximized).

More Information:

To avoid performing code in the Form_Resize event for this extra Resize event, you can set a global variable in the form's Unload event procedure that you can check in the Resize event procedure to determine if a normal Resize or an extra Resize event is occurring.

To work around this problem:

1. Dimension a flag variable in the General - Declarations section of the form in question:

```
Dim ExtraEventFlag%
```

2. Set the flag in the Form_Unload event:

```
Sub Form_Unload (Cancel as Integer)
    ExtraEventFlag% = -1
End Sub
```

3. Check the flag variable's value before performing code in the Resize event:

```
Sub Form_Resize
    If Not ExtraEventFlag% Then
        '** or other code for the Resize event..
        Beep
    End If
End Sub
```

The Form_Unload event should occur before the Form_Resize event; it should also set the flag variable and prevent the code for the Resize event from executing for this extra event.

Steps to Reproduce Problem

1. Start Visual Basic, or from the File menu, choose New Project.

2. In Form_Resize, add a Beep statement.

3. Set Form1.WindowState = 1 (Minimized).

4. Run the program. One beep occurs, and the minimized icon appears at the bottom of the screen.

5. Double-click the icon to restore the form. A beep occurs, indicating that a resize event has occurred.

6. Double-click the Control menu of Form1 to close the form. Another beep occurs, indicating that a resize event has occurred.

For forms with a Normal WindowState, no Resize event occurs when you close the form.

Text Too Narrow with Italic Fonts in Visual Basic Labels

Article Number: Q79117
Modified: 20-DEC-1991

Summary:

When the FontItalic property of a label control is set to True (-1), the text in the caption property is incorrectly formatted into lines that are about half as wide as the label width.

When the FontItalic property is set to False (0), the text is correctly formatted so that each line of text correctly occupies the width of the label.

More Information:

To work around this problem, use any one of the following alternatives:

1. Make the label wider so that the text appears as desired.

2. Set the AutoSize property on the label to True (-1). Note that this will format the label caption into one line of text.

3. Create several labels, one for each line of text.

4. Set the FontItalic property to False (0).

Steps to Reproduce Problem

1. Start Visual Basic. From the File menu, choose New Project (ALT, F, N). Form1 will be created by default.

2. Place a label (Label1) on Form1.

3. Set the label's Caption property to "This is a line of text" (without the quotation marks).

4. Make the label just wide enough to display the entire caption on one line. Make the label tall enough to display three or four lines of text.

4. Set the FontItalic property to True. The caption is incorrectly formatted as two lines.

If you further reduce the width of the label, it incorrectly formats the lines to about half the new width of the label.

Documentation Issues

Visual Basic comes with the an excellent set of manuals for the person who is starting to learn Visual Basic for the first time. The "Programmers Guide" addresses the look and feel of Visual Basic. The "Language Reference" includes a list in alphabetical order the methods, events, functions, and Sub procedures. There are also additional references that the motivated Visual Basic programmer may want to read.

Undocumented Separator Property of a VB Menu Item

Corrections for Errors in Visual Basic Version 1.0 Manuals

Visual Basic Online Help Example Errors

Incorrect Jumps in Visual Basic Online Help "How To" Section

Error Messages

Visual Basic catches many errors that creep into your code. However, not all the error messages may be self-explanatory. These articles discuss some error messages that may be puzzling when they are first encountered.

VB "Out of Stack Space" with LoadPicture in Form_Paint Event

"Method Not Applicable for This Object" with SetFocus

"Method Not Applicable..." with IsHidden Method in VB

Some Invalid DrawMode Values Return 1-16 Instead of Error

"Overflow" in VB Drawing Circle Segment with Radius of Zero

"Overflow" Printing Too Large a String to Form or Printer

"Unresolved External" Attempting to Use CDK VBDirtyForm()

VB AddItem Method Gives "Illegal Function Call" for List Box

VB Help Misleading Error, "Unable to Find Windows HELP.EXE"

VB Run-Time .EXE Error "Must Have Startup Form or Sub Main()"

FormName Not in Correct Order After "Out of Memory" Error

In VB, Clipboard.SetData Gives "Invalid Format" with Icon

"Out of Memory" May Lead to Unstable Visual Basic Environment

VB SETUP.EXE "Insufficient Disk Space on: C:\WINDOWS"

VB "Illegal Function Call" Under Windows Low Memory Condition

CDK Issues

The Visual Basic Control Development Kit (CDK) allows you to write custom control files. The creation of custom controls is the most powerful kind of extensibility in Visual Basic. When you are writing a custom control, you are in effect writing underlying code in Visual Basic itself.

EVENTINFO Topic Correction for VB CDK VBAPI.HLP File

Documentation Errors in First Printing of VB CDK Guide

How to Use CodeView for Windows (CVW.EXE) with Visual Basic

Declare Currency Type to Be Double When Returning from DLL

Huge Array Support in DLL for Visual Basic for Windows

In VB CDK, LibInit Does Not Allow Local Memory Allocation

VB CDK VBAPI.LIB Contains CodeView Information; Alternative

How to Subclass a VB Form Using VB CDK Custom Control

VB CDK Custom Property Name Cannot Start with Numeric

Design Issues and Limits

Visual Basic was written with certain design specifications and limits. The articles below address these design specifications of Visual Basic.

Overlapping Controls Not Supported in Visual Basic

Comments and Blank Lines Increase Size of VB 1.0 .EXE File

Cannot Tile or Cascade Programs Created with Visual Basic

How to Optimize Size and Speed of Visual Basic Applications

Visual Basic Applications Cannot Act as Windows Shell

Control Overlaid by Another Control Fails to Refresh if Moved

Memory Limits in Visual Basic for Windows

Service Issues

Why Cooper Software Is Listed in Visual Basic's Copyright

Sales Specification for Visual Basic Available

List of Visual Basic Third-Party Add-On Products

Undocumented Separator Property of a VB Menu Item

Article Number: Q76550

Summary:

Microsoft Visual Basic contains an undocumented property for menu items that toggle between the caption and a separator bar. The Separator property is not documented in Visual Basic's manuals or online Help.

You should not use the Separator property in your Visual Basic applications because it may be removed from future versions of the Visual Basic language.

Microsoft did not intend to leave the Separator property in the Visual Basic product.

More Information:

Separator Property (Applies to Menu Items)

Description: Denotes if a menu item is a separator bar. Prevents or allows the value of the Caption property of the menu item to be displayed versus a separator bar.

Usage: [*form.*][*menuitem.*]Separator[= *boolean%*]

Remarks: The Separator property settings are as follows:

Setting	Description
True (-1)	Disables the display of the value of the Caption property as the menu item. Instead, a separator bar is displayed.
False (0)	(Default) for all other menu items. The menu item will display the value of the Caption property for that item.

When a menu item is created in the menu design window, the value of its separator property defaults to 0 unless the caption of that menu item is set to "-", then the Separator property defaults to -1 and a separator bar is displayed for that item in the menu instead of the value of the Caption property.

Corrections for Errors in Visual Basic Version 1.0 Manuals

Article Number: Q73655

Summary:

Below are corrections for documentation errors in the manuals shipped with Microsoft Visual Basic version 1.0 for Windows.

This master list of corrections includes and adds to the correction list found in sections 2 and 3 of the README.TXT file shipped with Visual Basic 1.0. Please use the article below as your master list for making corrections to the Visual Basic manuals.

More Information:

Microsoft Visual Basic comes with the following two manuals:

"Microsoft Visual Basic: Programmer's Guide"
"Microsoft Visual Basic: Language Reference"

In both manuals, all references to OS/2 and Presentation Manager (PM) are misprints and should be ignored.

Corrections to the "Microsoft Visual Basic: Programmer's Guide"

(page xv) In the middle of the page,
OpenError
should read
OpenError:

(page 61) In "To set the Caption property:", the last line of #2
"...you can skip Step 4 and proceed to Step 5."
should read
"...you can skip Step 3 and proceed to Step 4."

(page 76) Add the following at the end of the first paragraph:
"...to distribute copies of the run-time file; however, you can only distribute this run-time DLL with the .EXE file you have created in Visual Basic. You cannot distribute VBRUN100.DLL by itself."

(page 88) In the second line,
"...so that Ardvark < Zypher evaluates..."
should read
"...so that "Ardvark" < "Zypher" evaluates..."

(page 97) In the first line,
"...calls to the LogB procedure..."
should read
"...calls to the LogF procedure..."

(page 106) Code at bottom of page should read:
x = Shell("\winword\winword.exe c:\winword\plan.doc", 1)
and
y = Shell("\winproj\winproj.exe c:\winproj\schedule.wpr", 1)

(page 118) In the event procedure AddApp_Click, after the line of code Load AppName(LMenu), insert the following:

AppName(LMenu).Separator = 0

Note that Separator is a property of a menu item that is not documented in the manuals or in the on-line help. The Separator property denotes whether or not a particular menu item is or is not a Separator item. The caption of a Separator item in a menu cannot be changed.

(page 129) In the first line of the last paragraph,
"...tabbing to the button and..."
should read
"...tabbing to the option button group, using the arrow keys, and..."

(**page 131)** In the middle of the page,
"Oct$, Str$, or Hex$ function..."
should read
"Oct$, Format$, or Hex$ function..."

(page 146) Near the top:
MsgBox msg$[,type%[,type%[,title$]]
should read
MsgBox msg$[,*type%*[,*title$*]]

(page 153) In lines 8 through 14, all references to
CurrentX
should be
CurrentY and vice versa.

(page 170) Code, middle of the page:
TextBoxes(I).Visible = -1
should read
TextBoxes(I).Visible = 0
Add the following as the first line in the Form_Load procedure at the bottom of the page:
Form1.Show
and change the following:
Text1(I).Text = "Text1(" + Format(I) + ")"
to
Text1(I).Text = "Text1(" + Format$(I) + ")"

(page 182) Code at top of page:
[object.]Line[(x1, y1)] - [x2, y2) [, color]
should read
[object.]Line[(x1, y1)] - (x2, y2) [, color]

(page 194) In the middle of the page,
"...given in the reverse order (Pi/3, Pi/2)..."
should read
"...given in the reverse order (-Pi/3, -Pi/2)..."

(page 204) In the last paragraph, the second line
"...continuous line when mouse button..."
should read
"...continuous line when the left mouse button..."

(page 223) The second line in the last paragraph
"...(named GroupChoice)..."
should read
"...(named GroupList)..."

(page 248) All references to the property DrawColor should be omitted. There is no DrawColor property.

(page 261) In the first paragraph, the second line:
"...variable-length strings."
should read
"...fixed-length strings."

(page 267) In the code at bottom of page, all references to "Picture1.Picture" should read "Form1.Picture1.Picture"

(page 270) Code at bottom of page, third line from bottom:
Mid$(MyTime, 1, 2)...
should read
Mid$(MyTime, ((Hours-12)<10)+2, 2)...

(page 276) Third paragraph, second line:
"KeyPress event ....The same thing happens when..."
should read
"KeyDown event ...The KeyDown event gets the same code when..."

(page 288) In the last line,
"MB_EXCLAIM"
should read
"MB_ICONEXCLAMATION"

(page 289) In the code, all references to
MB_EXCLAIM
should be changed to
MB_ICONEXCLAMATION

(page 308) Third paragraph of code, second nested IF THEN statement:
If FileListBox.List(ind) = ...
should read
If FileListBox.List(Ind%) = ...

(page 311) The code under FileListBox_DblClick
FileListBox.Pattern = *.*
should read
FileListBox.Pattern = "*.*"

(page 313) Do Loop should read:
```
Do Until Instr(LastOne% + 1, Test$, "\") = 0
    LastOne% = Instr(LastOne% + 1, Test$, "\")
Loop
```

(page 320) Last paragraph, last sentence should read:
"The maximum length of a file that can be edited with Text Editor is a little less than 32,000 bytes, because that is the default maximum number of characters you can assign to a Visual Basic multiline text box control."

(page 323) Code, middle of the page:
```
Global Const ErrBadFileNameOrNumber = 52...
.
.
```

.
Global Const MB_EXCLAIM = 48...
should read
Global Const Err_BadFileNameOrNumber = 52...
.
.
.
Global Const MB_ICONEXCLAMATION = 48...

(page 330) Code sample seventh to last line and **(page 331)** second line:
EndLine$ = Input$(1, 1)
should read
EndLine$ = Input$(1, FileNum)

(page 339) Add the following as the first line after Sub Form_Load:
Form1.Show
Without this change, running the code will result in the error "Illegal Function call" on the statement FieldBoxes(0).SetFocus on the second-to-last line. This is because the Form is not yet visible.

(page 339) Code at bottom of page:
WorkingFileNum = FileOpener...
should read
WorkingFileNum% = FileOpener...

(page 341) All reference to FieldBoxes in the code example on this page should specify the form. For example:
Form1.FieldBoxes

(page 357) The last two sentences of the Note should be deleted and should be replaced with the following:
"If no application responds, Visual Basic generates a run-time error."

(page 360) In the Note at bottom of page, references to the ALT key should be ESC key.

(page 364) The Link Execute Event, second argument, last line of description:
"...the client receives a negative argument."
should read
"...the client receives a negative acknowledgment."

(page 369) Under the "Starting Other Applications" section:

"If you attempt to establish a DDE conversation with an application that is not currently running, Visual Basic displays a message asking if the user wants to start the application."
This should be changed to read as follows:
"If you attempt to establish a DDE conversation with an application that is not currently running, Visual Basic displays the message 'No foreign application responded to DDE request'."

In the program example, add the following as the first line after the Sub statement:
Const DDE_NO_APP = 282
Also, in the program example:
If Err = DDE_NO_APP
should be changed to read as follows:
If Err = DDE_NO_APP Then

(page 370) In "Requesting Data from Other Applications", all references to
"warm link"
should be references to
"cold link".

(page 371) In the first paragraph of "Notifying Other Applications When Data Has Changed,"
"(in the case of a warm link)"
should read
"(in the case of a cold link)".

(page 378) In "To load a custom control file:", delete step 2 (the second line). "Type or select the name..." should be step 2.

(page 380) Under "Special Considerations When Declaring DLL Routines", the statement

"The Visual Basic package contains a file that includes the declarations for all useful routines in the operation..."
is incorrect. This file is not included with Visual Basic, but is part of the add-on kit, "Microsoft Windows Programmer's Reference" Book and OnLine Resource, available from Microsoft at a charge.

(page 386) In the code at top of page, the first set of quotation marks around "Microsoft Excel" should be straight quotation marks.

Also, add the following paragraph after first code fragment:
"The use of ByVal when passing a string is necessary because the data type of that argument was declared As Any. Including ByVal when passing a string declaring As Any causes Visual Basic to convert the string to the null-terminated form expected by most DLL routines."

The following code at the bottom of the page:
Lib "User"...
should read
Lib "GDI"...

(Index, starting on page 423)

A "ALT key":
Delete "interrupting DDE 360"

"As Any":
Add to page list: "386"

C After "CurrentY":
Add the following two entries:
"Cursor Search Help files for Mouse Pointer"
"Cursor position see Insertion point"

E After "Errors":
Add new topic: "ESC Key, interrupting DDE 360"

I "Index property":
Add to page list on "creating control arrays": "117"

"Insertion point":

After "defined", add "locating 264"

L "Line Method":
"boxes 188-188" should read "boxes 188-189"

M "Microsoft Visual Basic":
Change page list on "starting programs" from "17-18" to "16-17"

S "Strings":
"variable-length 251, 385" should read "use of ByVal with variable-length 251, 385, 386"

Corrections to "Microsoft Visual Basic: Language Reference"

(page 6) Under the column Action in the category Procedures,
"Define a Sub procedure"
should read
"Define a procedure."
Also,
"Call a procedure"
should read
"Call a Sub procedure"

(page 9) In the Property column and Windows category,
"hWin"
should read
"hWnd"

(page 11) The Height property for the Screen object is Read-only at run time for all objects (and should be marked in this chart with half-dots instead of whole dots).

(page 27) Add to the Note the following paragraph:
"When you minimize a form whose AutoRedraw Property is set to False(0), ScaleHeight and ScaleWidth are set to icon size. When AutoRedraw is set to True (-1), ScaleHeight and ScaleWidth remain the size of the restored window."

(page 31) For the Description:
"...for an object; for forms and picture boxes..."
should read
"...for an object, for forms and text boxes..."

For the Note:
"...use the Show method with the form at run time. To make a form modeless, set its Visible property to True(-1)."
should read
"...use the Show method with Style% = 1. To make a form modeless, use Show with Style% = 0."

Add the following paragraph at the end of the Remarks section:
"Because of appearance, the BorderStyle for forms with a menu can only be set to Sizeable (2) or Fixed Single (1). Setting the BorderStyle property to None (0) or Fixed Double (3) forces the BorderStyle property to Fixed Single (1)."

(page 52) The graphic image shown is an example of a check box; it should show a combo box example.

(page 147) In the Description, "read-only" should read "read-write"

Change the Note to read as follows:
"For a form icon to be functional, the BorderStyle property must be set to either 1 (Fixed Single) or 2 (Sizeable). The MinButton property must be set to True (-1)."

"At run time, you can assign an object's Icon property to another object's DragIcon or Icon property. You can also assign an icon returned by the LoadPicture function. Doing this assigns an empty (null) icon, which enables you to draw on the icon at run time."

(page 154) Despite references on page 154 and in the VB.EXE OnLine Help, the INPUT$ function is not supported for files opened with random access. Attempting to use INPUT$ on a file opened for random access results in a "Bad file mode" error message. (Use of INPUT$ on random files was eliminated in Visual Basic, as was the older Basic statement FIELD.)

(page 156) Under Condition,
"strexpr1$ found in strexpr2$"
should read
"strexpr2$ found in strexpr1$"

(page 158) Near the bottom in the Remarks section:
"...property determines if timer responds..."
should read
"...property determines if the timer control responds..."

(page 175) In the Remarks section, add the following at the end of the "Cancel" description:
"(The default is set to -1, meaning cancel)"

(page 184) In the second line of the Note,
"...by pressing the ALT key."
should read
"...by pressing the ESC key."

(page 188) Last line of description should read:
"The List property is not available at design time; it is read-only or drive, file, and directory list boxes and read-write for combo and list boxes."

(page 204) For the Syntax:
"Mid$ (stringexpression$, start&, [length&])"
should read
"Mid$ (stringexpression$, start& [,length&])"

(page 209) In the Case statements of the Example, delete all numbers greater than 12.

(page 244) This property (Pointer property) does not exist. The name was changed to MousePointer Property (page 214).

(page 285) The syntax for ClipBoard.SetData should be changed from
"ClipBoard.SetData (*data%*, [*format%*])
to
"Clipboard.SetData *data%*, [*format%*]"

(page 290) Middle of the page:
"...further user input can occur."
should read
"...other forms can respond to events or accept user input."

(Appendix A, page 337) ANSI Table at bottom of page:

"Values 8, 9, 10, and 13 convert to tab, backspace, linefeed..."

should read

"Values 8, 9, 10, and 13 convert to backspace, tab, linefeed..."

VB "Out of Stack Space" with LoadPicture in Form_Paint Event

Article Number: Q72675

Summary:

An "Out of stack space" error can occur when you use a LoadPicture method within a Form_Paint event.

More Information:

The Visual Basic stack can be exhausted when the LoadPicture method is executed within a Paint event. The LoadPicture method generates a Paint event itself, and when performed within a Paint event, the program will repeat the cycle until the stack is exhausted.

The following code example demonstrates that the Form_Paint event is a recursive procedure when a LoadPicture method is included in the Paint event code.

After you add the code to your program, run the program and note how many times the message "Form_Paint Count :" is displayed within the Immediate Window before you receive the "Out of stack space" error message.

```
Sub Form_Paint ()
    Static Count
    Count = Count + 1
    Debug.Print "Form_Paint Count : "; Count
    Form1.Picture = LoadPicture("c:\windows\chess.bmp")
End Sub
```

To remedy the situation, move LoadPicture to another event handler, such as the Form_Load event. Since these bitmaps are automatically refreshed when needed, you don't have to maintain the picture within a Paint event.

The Visual Basic stack is limited to 16K and cannot be changed.

"Method Not Applicable For This Object" with SetFocus

Article Number: Q72676

Summary:

Visual Basic will return "Method Not Applicable For This Object" whenever you use the SetFocus method on a control that is on a form that is not visible.

More Information:

A control must be visible before you can execute the SetFocus method to that control. To make a control visible, the form must be loaded and the Visible property for the control must also be True {-1}.

For example, create two forms, and on each form, create a command button.

In the code segment that follows, Form2 is not loaded yet, so you must show Form2 before you can set the focus to Form2.Command1. Once Form2 has been loaded, you can set the focus to the visible control Form2.Command1. When you click the Form1.Command1 twice, you will receive the error "Method Not Applicable for This Object" because Form2.Command1's Visible property was set to False during the previous call to Form_Click.

This is the load event for Form1:

```
Sub Form_Load ()
    Form2.Show
      'this is the default setting
    Form2.Command1.Visible = 0
End Sub
```

This is the click event for Form1.Command1:

```
Sub Command1_Click ()
    Form2.Command1.SetFocus
      'hide the control
    Form2.Command1.Visible = 0
End Sub
```

"Method Not Applicable..." with IsHidden Method in VB

Article Number: Q73154

Summary:

Using the word "IsHidden" as the name for a variable, Sub, Function, or object incorrectly gives the following error at run time in Visual Basic:

Method not applicable for this Object

To work around this problem, avoid using the name "IsHidden".

More Information:

Steps to Reproduce Problem

1. Within any event procedure, try the following:

```
Sub Form_Click()
    Print IsHidden
End Sub
```

2. This gives a syntax error asking you to add a "(" to the method, so continue by adding "()":

```
Sub Form_Click()
    Print IsHidden()
End Sub
```

3. This results in the following format after pressing ENTER after IsHidden():

```
Sub Form_Click()
    Print IsHidden()
Sub End
```

This is the format for a predefined method called IsHidden().

After following the steps above, you will receive a run-time error "Method not applicable with this Object."

Some Invalid DrawMode Values Return 1-16 Instead of Error

Article Number: Q75174

Summary:

The Microsoft Visual Basic DrawMode property will not give an error for some invalid values (for example, numbers in the range 257 to 272, 513 to 528, and so forth, and for negative numbers in the range -240 to -255, -496 to -511, and so forth).

More Information:

The code below demonstrates the problem. When you run this program, the DrawMode is assigned to 257, and when the DrawMode is printed, it prints 1, showing that the values are mapping onto 1 through 16.

Example

1. In the VB.EXE environment, create a New Form.

2. In the Form_GotFocus event procedure, add the following lines of code:

```
DrawMode = 257
Print DrawMode
```

3. Run the program (press F5). The program will incorrectly display a 1 on the form, instead of giving the error message "Illegal Property Value."

By analyzing the range of values that map onto 1 through 16, any number that satisfies the following equation will not generate an error, but will instead give values in the range of 1 to 16

$$Number = (x * 256) + y$$

where *x* is a whole number between -128 and 127, and *y* is between 1 and 16 inclusive (that is, any number in the range of a multiple of 256 plus 1 to 16 will map onto 1 through 16).

"Overflow" in VB Drawing Circle Segment with Radius of Zero

Article Number: Q73280

Summary:

When using the Microsoft Visual Basic version 1.0 Circle statement to draw a segment of a circle with a radius of 0, an "Overflow" error incorrectly occurs.

More Information:

The following statement demonstrates the problem:

```
Circle (0,0), 0,, 4, 5
```

When you run the above statement, an "Overflow" error incorrectly occurs.

In contrast, using the Circle statement to draw an entire circle of radius 0 works correctly without an error (correctly drawing nothing); for example:

```
Circle (0,0), 0
```

"Overflow" Printing Too Large a String to Form or Printer

Article Number: Q74517

Summary:

An "Overflow" error message may occur when you print a long string in Microsoft Visual Basic.

When a character is printed using the Print method, the CurrentX and CurrentY coordinates are also updated for the object being printed to. If the string being printed is long enough to cause the value of the CurrentX property to exceed 32,767 twips, an "Overflow" error will occur. This behavior is by design.

"Overflow" can be caused by printing a single long string or by repeatedly printing shorter strings that are appended to the end of the last string (using the Visual Basic semicolon [;] operator).

More Information:

Steps to Reproduce Problem

1. Start Visual Basic or choose New Project from the File menu.

2. Place a label control on Form1.

3. Add the following code to the Form_Click event:

```
Sub Form_Click()
    For index% = 1 to 1000
        Print "A";
        Label1.Caption = Str$(CurrentX)
    Next
End Sub
```

4. From the Run menu, choose Start.

5. Click on Form1.

An "Overflow" error will occur. You can examine the label caption to see that the value of Form1.CurrentX plus the TextWidth of "A" exceeded 32,767 at the time of the error.

"Unresolved External" Attempting to Use CDK VBDirtyForm()

Article Number: Q74338

Summary:

The VBAPI routine **VBDirtyForm**() is documented within Visual Basic's add-on Control Development Kit (CDK) and is defined with the CDK Header file, VBAPI.H. However, the LINK.EXE error "Unresolved external" occurs when linking your dynamic-link library (DLL) if you attempt to use this function, because **VBDirtyForm**() does not actually exist.

More Information:

The functionality of **VBDirtyForm**() can easily be achieved using **VBSetControlProperty**() to assign a value to any property that has been defined with the property flags of PF_fSaveData and/or PF_fSaveMsg.

When a property is assigned a value that was defined with one or both of the property flags, PF_fSaveData or PF_fSaveMsg, VB.EXE takes care of writing the property value to disk when the form is saved, and is made aware of any changes made to any property of any control or the form itself. Thus, VB.EXE knows to save the form when exiting or opening a new project.

VB AddItem Method Gives "Illegal Function Call" for List Box

Article Number: Q75642

Summary:

Below is an example of a problem with adding an item to the list box object in a Visual Basic program.

More Information:

Steps to Reproduce Problem

1. Start Visual Basic with a New Project.

2. Add a list box and a command button on Form1.

3. Add the following code in the Command1_Click event procedure:

```
Sub Command1_Click ()
    a$ = "hello"
    List1.AddItem a$, 1
End Sub
```

4. Run the program by pressing the F5 key. The error message "Illegal function call" should be displayed. Press the F1 key for additional help on this error. Help will explain about an improper or out-of-range argument on the function.

The actual problem concerns the index% of the AddItem method, discussed on page 21 of the "Microsoft Visual Basic: Language Reference" for version 1.0.

Workaround

The following are two different ways to resolve the problem:

Leave out the index% option on the AddItem method as shown:

```
Sub Command1_Click ()
    a$ = "hello"
    List1.AddItem a$
End Sub
```

Start out with the index% option as 0 instead of 1 as shown:

```
Sub Command1_Click ()
    a$ = "hello"
    List1.AddItem a$, 0
End Sub
```

The addition of Option Base 1 to the General section of the Form's Code window does not eliminate the above problem.

VB Help Misleading Error, "Unable to Find Windows HELP.EXE"

Article Number: Q76549

Summary:

When Windows system resources are low (less than five percent free), invoking Visual Basic online Help may display a misleading error dialog box, such as "Unable to find Windows HELP.EXE."

More Information:

Steps to Reproduce Problem

1. Start Visual Basic.

2. Check Windows free resources (choose About from the Windows Help menu). If free resources are less than five percent, proceed to step 4.

3. Start another Windows application. Go to step 2.

4. From the Visual Basic OnLine Help, choose the Index button.

This problem, while possibly misleading, is not destructive in any way. To regain access to the Visual Basic Help system, you must close applications until you have more than five percent of free system resources.

VB Run-Time .EXE Error "Must Have Startup Form or Sub Main()"

Article Number: Q76634

Summary:

The Make EXE File command on the File menu creates an executable file regardless of whether or not you have chosen a startup form or Sub Main.

An executable created without a startup form or Sub Main will generate the error message "Must have Startup form or Sub Main()" at run time.

To work around the problem, make sure you have chosen a startup form or have a Sub Main procedure. To choose a startup form, you can use the Set Startup Form command on the Run menu.

More Information:

Steps to Reproduce Problem

1. Run Visual Basic.

2. From the File menu, choose Remove File.

3. From the File menu, choose Add Form. At this point, there is no startup form selected.

4. From the File menu, choose Make EXE File. The .EXE file is created, but it has no startup form or Sub Main. VB.EXE fails to give you an error message at this point.

5. Exit Visual Basic and run the .EXE file from the Windows Program Manager.

Running the .EXE file causes the error message "Must have startup form or Sub Main()."

If you choose the Start command from the Run menu, you will be prompted for a startup form or Sub Main.

FormName Not in Correct Order After "Out of Memory" Error

Article Number: Q76983

Summary:

If, when creating a new form, you receive an "Out of memory" error message, no form will be loaded. However, the default FormName is still incremented by 1 so that when a new form can be created (for example, by deleting an already existing form) after getting the error, the FormName of the newly created form is not in the correct sequence.

More Information:

Steps to Reproduce Problem

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

2. Create new forms by choosing New Form from the File menu (ALT, F, F) until you get an "Out of Memory" error (on one machine, this occurred when trying to load Form52).

3. Choose the OK button to acknowledge the error message.

4. Delete Form51 by choosing Remove File from the File menu (ALT, F, R).

5. Create one more form. In the example mentioned in step 2, Form53 was the next form, even though Form52 was never created.

EVENTINFO Topic Correction for VB CDK VBAPI File

Article Number: Q74649

Summary:

Below is a correction to the VBAPI.HLP file provided in the Microsoft Visual Basic Control Development Kit (CDK) (add-on kit number 046-050-022).

In the VBAPI.HLP file, for the structure topic named EVENTINFO, a column title is incorrectly labeled "Flag Value"; it should instead be labeled "Field Name."

Documentation Errors in First Printing of VB CDK Guide

Article Number: Q74676

Summary:

Below is a list of documentation errors in the first printing of the "Microsoft Visual Basic: Control Development Guide" shipped with the Visual Basic Control Development Kit(CDK).

The documentation errors listed below may not apply to later printings of the "Microsoft Visual Basic: Control Development Guide."

Many of the errors were corrected in the CDK Help file before shipping.

The CDK is an add-on to the Microsoft Visual Basic programming system version 1.0 for Windows.

More Information:

Corrections to the "Microsoft Visual Basic: Control Development Guide"

(page 16) Important:

"...table must list properties in..."
should read
"...table must list events in..."

(page 79) Middle of the page:

"...fourth field in the EVENTINFO structure..."
should read
"...fourth field in the PROPINFO structure..."

Code at bottom of page:
"OFFSETIN(MYCTL, Language),"
should read
"OFFSETIN(MYCTL, enumLanguage),"

(page 82) Bottom of the page, third bullet:

"...respond to VBM_SAVEPROPERTY..."
should read
"...respond to VBM_LOADPROPERTY..."

(page 112) VBXPixelsToTwips, VBYPixelsToTwips:

Return data type is listed in Syntax as SHORT; it should be LONG.
"...in logical twips into a measurement in pixels..."
should read
"...in logical pixels into a measurement in twips..."
In the chart, Type should list as SHORT.

(page 136) In the table, the incorrect heading

"Flag Value"
should be changed to
"Field Name".

How to Use CodeView for Windows (CVW.EXE) with Visual Basic

Article Number: Q75612

Summary:

You can use CodeView for Windows (CVW) to debug a dynamic-link library (DLL) or custom control that is called from Visual Basic.

More Information:

CVW is distributed with the Microsoft Windows Software Development Kit (SDK). CVW can be a useful tool for debugging DLLs and custom controls written for a Visual Basic program.

CVW takes the following command-line arguments:

```
[path]CVW.EXE /L [dynamic link library] [executable program]
```

where:

[dynamic library] is your DLL or custom control.
[executable program] is the EXE that uses your DLL/custom control.

The "/L" will tell CVW that this is a DLL or custom control.

You can invoke CVW from the Windows Program Manager in any of the following ways:

1. From the Windows Program Manager File menu, choose New, and specify CVW.EXE as a Program Item with proper arguments. You can then double-click on the CVW icon to run CVW.EXE.

2. From the Windows Program Manager File menu, choose Run, and enter CVW.EXE and its command-line arguments.

3. Invoke CVW with no arguments, and at the prompts, enter the program name and DLL/VBX that you want to debug.

The example below demonstrates how to invoke CIRCLE3.VBX, which comes with the Microsoft Visual Basic Control Development Kit (CDK):

1. Run CVW.EXE from the Program Manager as follows.

```
[path]CVW.EXE /L [path]CIRCLE3.VBX [path]VB.EXE
```

Note: You can just specify an .EXE program that was written in the Visual Basic environment instead of specifying the VB.EXE environment itself. If you do this, skip steps 7, 8, and 9 below.

Note: If you invoke CVW.EXE with no command-line arguments, CVW.EXE will prompt you for command-line arguments. Specify the VB.EXE file that uses the *.VBX file. CVW.EXE will then prompt you for "Additional DLLs...". Specify the *.VBX file at this prompt. Skip to step 4.

2. When CVW is loaded into the debug monitor, the following message will be displayed:

```
No Symbolic information for VB.EXE
```

3. Load the Visual Basic program by choosing the OK button.

4. Load the CIRCLE3.VBX source code by choosing File - Open Module from the menu. You should see a list of "c" source code in the list box. Select CIRCLE.C, which corresponds to the CIRCLE3.VBX source code.

5. Locate the WM_LBUTTONDBLCLK message and set a breakpoint on the first "IF" statement.

6. Press F5 (Run) to go to your Visual Basic program.

7. Select Add File and add the CIRCLE3.VBX file. You should see a button with a circle added to the Toolbox window.

8. Select the custom control from the Toolbox window and add it to your form.

9. Press F5 to run your program.

10. Double-click the circle. When your breakpoint is encountered, focus will be set to CVW and execution will stop at your breakpoint. You can now step through your program.

11. Press F5 to return to the Visual Basic program.

References

Declare Currency Type to Be Double When Returning from DLL

Article Number: Q72274

Summary:

When using Visual Basic, 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(...);
```

References

Huge Array Support in DLL for Visual Basic for Windows

Article Number: Q72585

Summary:

A dynamic-link library (DLL) is available that contains functions for managing arrays larger than 64K from Microsoft Visual Basic version 1.0 for Windows. This DLL also provides the ability to create arrays with more than 32,767 (32K) elements per dimension, and to redimension arrays while preserving the data inside the arrays.

This file can be found in the Software/Data Library by searching for the filename BV0442, the Q number of this article, or S13082. BV0442 was archived using the PKware file-compression utility. When you decompress BV0442, you will obtain the following files:

HUGEARR.DLL, HUGEARR.BAS, HUGEARR.C, HUGEARR.DEF, HUGEARR.H, HUGEARR.TXT, MAKEFILE

These files are also available on disk in the application note "Huge Array Support in DLL for Visual Basic for Windows" (BV0442). This application note can be obtained by calling Microsoft Product Support Services.

This information applies to Microsoft Visual Basic programming system version 1.0 for Microsoft Windows, and to Microsoft Windows 3.0 Software Development Kit (SDK). HUGEARR.DLL is provided only as an example, which you are free to modify, and Microsoft makes no performance or support claims for HUGEARR.DLL or its associated files.

More Information:

To use the functions in HUGEARR.DLL, copy the declarations contained in HUGEARR.BAS into your global module in Visual Basic and copy HUGEARR.DLL to your Windows directory. The functions can then be used like any other Windows DLL functions.

HUGEARR.DLL allocates memory using the Windows API function **GlobalAlloc**(). This means that the largest array that can be allocated is 1 MB in standard mode, and 64 MB in 386 enhanced mode for Windows.

The following routines are contained in HUGEARR.DLL. For a complete description of the parameters and/or return values of these routines, see Visual Basic's Declare statement for the routine in question in the file HUGEARR.BAS. For additional notes on using these functions, see the HUGEARR.TXT reference file.

1. HugeDim: Dimensions an array and returns a handle to that array.

2. HugeErase: Erases an array that was previously dimensioned using HugeDim.

3. HugeRedim: Redimensions an array created with HugeDim to a different size.

4. GetHugeEl, SetHugeEl: Gets or sets the contents of the specified array element in a given huge array.

5. HugeInt, HugeLong, HugeSingle, HugeDouble, HugeCurrency: Functions that return a value from a specific element in a huge array of the type corresponding to the function name (Integer, Long, Single, Double, or Currency data type.)

6. HugeUbound: Returns the upper bound of a given huge array.

7. NumHugeArrays: Returns the number of free huge array handles available.

HUGEARR.DLL is written in Microsoft C, and the C source code is provided in HUGEARR.C and HUGEARR.H. Advanced programmers can optionally modify and rebuild HUGEARR.DLL by using the Microsoft C Compiler version 6.0 or 6.0a and DLL libraries from the Microsoft Windows 3.0 Software Development Kit (SDK), and by running NMAKE.EXE with the enclosed MAKEFILE. The MAKEFILE tells LINK.EXE to use the enclosed linker definition file, HUGEARR.DEF.

References

Overlapping Controls Not Supported in Visual Basic

Article Number: Q73651

Summary:

Overlapping Visual Basic controls may not respond as expected to mouse events. For example, the bottom control will receive the mouse event even when it appears that you have selected the top control. The use of overlapping controls is not supported in Microsoft Visual Basic version 1.0.

More Information:

Although the Visual Basic design editor allows you to overlap controls, when you run the application the region of the controls that overlap may not function as you would expect.

For example, if two Command buttons, Command1 and Command2, overlap so that Command1 is partially on top of Command2, when you select Command1 within the region of overlap you would expect a Click event to be issued for Command1. However, the Click event may occur on Command2 even though it is underneath Command1 in the overlapping region.

Comments and Blank Lines Increase Size of VB 1.0 .EXE File

Article Number: Q73697

Summary:

Each line containing blank space or a comment in any Code window of a Visual Basic application adds 2 bytes to the size of the compiled executable file (.EXE). This is not a problem with Visual Basic, but rather it is part of Visual Basic's design.

More Information:

The 2-byte overhead for each line containing blank space or a comment is generated as part of the pseudocode for the application in the VB.EXE development environment. The program is run in "interpreted mode" based on this pseudocode. Because an .EXE program is generated based on this pseudocode (in other words, Visual Basic does not use a compiler and linker), the 2-byte overhead is copied to the .EXE program. The only workaround for this behavior is to remove comments and blank lines before compiling the Visual Basic project.

Cannot Tile or Cascade Programs Created with Visual Basic

Article Number: Q73698

Summary:

Applications that have been created with Microsoft Visual Basic 1.0 do not tile or cascade as other Windows applications do.

More Information:

Visual Basic creates applications that are pop-up windows. This window style does not respond to the tile or cascade message sent from the Windows 3.0 Task List or other applications that support the cascade and tile features.

You can verify this action by launching two applications created in Visual Basic, then bringing up the Windows Task List by pressing CTRL+ESC, and from the Task List choosing either the Cascade or Tile button. Note that nothing has changed in the arrangements of these two Visual Basic application windows. You may have expected the Visual Basic application windows to cascade or tile as other Windows applications do, but they will not do so.

How to Optimize Size and Speed of Visual Basic Applications

Article Number: Q73798

Summary:

This article describes how to optimize size and speed for applications in Microsoft Visual Basic programming system version 1.0 for Windows.

More Information:

Below are guidelines to help increase speed, available resources, available RAM, and available disk space in Visual Basic:

Increase Speed

- Preload forms.
- Store graphics as bitmaps.
- Place debug routines in a separate module.
- Use dynamic-link library (DLL) routines.

Increase Available Resources

- Create simulated controls using graphic objects.
- Draw graphics images during run time.

Increase Available RAM

- Use Integer variables instead of Single variables.
- Create dynamic arrays in order to free arrays when not needed.
- Drop/unload controls and forms when not needed.
- Use local variables.

Increase Disk Space

- Build controls at load time.
- Minimize header size.
- Delete unnecessary functions and subroutines.
- Delete unused objects and associated methods.

Visual Basic Applications Cannot Act as Windows Shell

Article Number: Q73801

Summary:

An application created by Visual Basic cannot be used as the Windows shell. Attempting to run an application as the Windows shell results in a Windows "Unrecoverable Application Error" (UAE) message. This is expected behavior when attempting to run a Visual Basic application as the Windows shell. This is not a problem with Visual Basic, but rather a design limitation. This information applies only to Visual Basic .EXE programs.

More Information:

A user-defined shell application can be specified in the Windows system initialization (SYSTEM.INI) file. The default shell is PROGMAN.EXE, or the Program Manager. If a Visual Basic program is specified as the customized Windows shell, a Windows UAE will occur on an attempt to run Windows from the MS-DOS command line.

A Visual Basic application cannot be run as the Windows shell because it does not contain the special set of startup code required by a Windows shell application. The only way to create a Windows shell application is to use the C Compiler and the Windows Software Development Kit (SDK) to write a non-Visual Basic application.

Steps to Reproduce Problem

Warning: The following steps require changing the Windows system initialization file (SYSTEM.INI) in a manner such that Windows will not run successfully unless the file is restored from MS-DOS. The file can be restored from MS-DOS by using a backup copy of the SYSTEM.INI file or by restoring the SYSTEM.INI file with a text editor from MS-DOS.

1. Start Visual Basic.

2. From the File menu, choose New Project.

3. From the File menu, choose Make .EXE File.

4. Choose the OK button to select PROJECT1.EXE as the .EXE filename.

5. Exit Visual Basic.

6. Start Windows Notepad.

7. From the File menu, choose Open.

8. In the Filename text box, type **c:\windows\system.ini** or the path and filename of the SYSTEM.INI file on your system.

9. Choose the OK button.

10. Change the line the reads "shell=progman.exe" to "shell=c:\vb\project1.exe", or the appropriate path to the file created in step 4 above.

11. From the File menu, choose Save.

12. Exit Notepad.

13. From the Windows Program Manager File menu, choose Exit (you should exit back to MS-DOS).

14. At the MS-DOS prompt, run Windows.

Warning: When you attempt to run Windows, a UAE will occur. You will need to reboot your computer and modify the SYSTEM.INI with a text editor from MS-DOS such that the line "shell = c:\vb\project1.exe" appearing in the SYSTEM.INI is changed back to "shell=progman.exe".

Control Overlaid by Another Control Fails to Refresh If Moved

Article Number: Q74519

Summary:

Visual Basic does not support overlapping controls. Having overlapping controls can result in portions of a control not refreshing correctly. If controls are moved over each other, one or both of the controls may not correctly refresh even when the controls are moved apart. This problem is known to occur when controls are resized at run time using the Move method or by changing the Height and Width properties as a result of a Form_Resize event. Because controls must be resized one at a time, it is possible that one control will briefly overlap another control during the resize process at run time. The control that was briefly overlapped may not refresh properly. An example of this behavior is given farther below.

This behavior can be improved by performing the Refresh method on every overlapping control at run time, after an overlapped control has been moved or after a form that contains overlapping controls has been resized.

This is not a problem with Visual Basic, but the nature of overlapping controls under Visual Basic. This behavior occurs at run time in the Visual Basic development environment or in a Visual Basic .EXE program.

More Information:

Steps to Reproduce Problem

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

2. Add a picture control (Picture1) to the default form (Form1).

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

4. Add a vertical scroll bar (VScroll1) to Form1.

5. Using the mouse, double-click Form1 to bring up the Code window.

6. Within the Resize event procedure of Form1, add the following code:

```
Sub Form_Resize ()
    Picture1.Move 0, 0, ScaleWidth - VScroll1.Width, ScaleHeight - Command1.Height
    VScroll1.Move ScaleWidth - VScroll1.Width, 0, VScroll1.Width, ScaleHeight - Command1.Height
    Command1.Move 0, ScaleHeight - Command1.Height, ScaleWidth, Command1.Height
End Sub
```

7. Run the program.

8. Using the mouse, resize the form by extending the bottom or right sides. When the bottom edge of the form is extended, the command button (Command1) will not refresh. When the right edge of Form1 is extended, the scroll bar will not refresh. The refresh problems are caused because Picture1 is expanded and temporarily overlaps the control. When the control (VScroll1 or Command1) is moved out of the way, it is not refreshed.

To work around this behavior, use the Refresh method for Picture1, VScroll1, and Command1 after the controls have be moved. Add the following statements to Sub Form_Resize (after the Command1.Move statement) above to overcome the behavior:

```
Picture1.Refresh
VScroll1.Refresh
Command1.Refresh
```

Memory Limits in Visual Basic for Windows

Article Number: Q72879

Summary:

The following memory limitations apply to Microsoft Visual Basic programming system version 1.0 for Windows.

Note: This information applies only when running the retail version of Microsoft Windows 3.0. Different memory limitations may apply when running Visual Basic under the debug version of Windows provided with the Microsoft Windows Software Development Kit (SDK).

More Information:

Each Function or Sub procedure can have up to 64K of pseudocode (p-code).

Each form, module, and global module gets its own data segment (up to 64K) for the allocation of all static data, strings, and simple variables (declared in the General section and Sub and Function procedures).

Each array of any data type gets its own data segment, up to 64K. Arrays larger than 64K cause a "Subscript out of range" error.

Huge arrays (arrays larger than 64K) are not directly supported in Visual Basic, but you can support huge arrays through the use of a Windows dynamic-link library (DLL). For more information about support for huge arrarys, search on the following words:

huge arrays

The properties for all controls on a form and the properties of the form itself go into a single data segment limited to 64K, except the following:

The List() property of a list box
The List() property of a combo box
The Text property of a text box

The List() property gets its own data segment, limited to 64K, for each list box and combo box. The Text property of a text box has a default size limit of 32K, which can be increased to 64K with a call to a Windows API function. For more information on how to increase the amount of text (from 32K to 64K) that can be entered into a text box, search on the following word:

EM_LIMITTEXT

Other memory limits relating to the properties of controls and forms include:

1. Each item in the List() property can be up to 1K; any item over this limit is truncated.

2. The Caption property of a control can be up to 1K; any caption over this limit is truncated.

3. The Tag property of a control can be up to 32K; any tag over 32K causes an "out of memory" error.

There is one name-and-symbol table up to 32K in size per form, module, or global module. A name-and-symbol table contains the actual text of Sub function and Sub procedure names, variable names, line numbers, line labels, and an additional 4 bytes of overhead for each of these names and symbols.

If the 32K size limit is exceeded for a name-and-symbol table, an "out of memory" error will occur. To solve this problem, break up the form or module into multiple forms or modules. note, this cannot be done with the global module. Only one global module is allowed; if you exhaust the global module's name-and-symbol table, there is no workaround other than to use shorter variable names.

The stack is 16K, with just one stack per application. The 16K stack size cannot be changed. Note that an "out of stack space" error can easily occur when your program uses uncontrolled recursion.

If you run Visual Basic on the debug version of Microsoft Windows provided with the Microsoft Windows SDK, all properties [including List() and Text properties] go into a single segment, up to 64K per form or module. Other memory-management limits may also differ under the debug version of Microsoft Windows. The debug version of Windows is created by copying a set of dynamic-link library (DLL) files from the Windows SDK into your C:\WINDOWS\SYSTEM subdirectory. These special DLLs perform additional error checking, including a check for stack overflow.

Why Cooper Software Is Listed in Visual Basic's Copyright

Article Number: Q72747

Summary:

The Microsoft Visual Basic copyright notice acknowledges Cooper Software in both the sign-on dialog box and in the About dialog box from the Help menu. Visual Basic uses technology from a forms engine purchased from Cooper Software. The acknowledgment in Visual Basic is part of the contract between Microsoft and Cooper Software.

Sales Specification for Visual Basic Available

Article Number: Q77906
MODIFIED: 2-DEC-1991

Summary:

You can obtain the following detailed sales literature by calling Microsoft End User Sales at (800) 426-9400:

- Visual Basic, Sales Specification (part number 098-22158)
- VB Library for SQL Server, Sales Specification

Removing Disk During VB Setup Terminates SETUP, Missing Files

Article Number: Q73157

Summary:

While running SETUP.EXE for Visual Basic, you may fail to copy all files (or experience other problems) if you try to remove the disk while the SETUP.EXE program is in progress.

To work around this problem, wait until SETUP indicates that 100 percent of the files are copied before removing the Visual Basic floppy disk.

More Information:

Below is an example of one specific problem:

When running SETUP.EXE on Disk 1 of Visual Basic (using 1.2 MB 5.25- inch disks), you can choose the option to install Visual Basic only. Setup's bar graph will start by displaying 4 percent done, and while the large file C:\VB\VB.EXE is being copied, the disk drive light may go off and you might assume that Visual Basic is finished running Setup. At this point you might (mistakenly) remove Disk 1. Then a message tells you that Setup is complete, and you can exit Setup.

You have now installed VB.EXE, but you did not install some of the other important files, such as VBRUN100.DLL, which is needed to run your compiled applications under Windows version 3.0.

If you want to install Visual Basic by selecting Visual Basic only, then you need to let Disk 1 complete its processing by waiting until the bar displays 100 percent. If you let Disk 1 run to completion, then the installed directory C:\VB should correctly contain the following files:

VB.EXE
VB.HLP
VBRUN100.DLL
README.TXT
PACKING.LST
SETUP.EXE
DECOMP.DLL
CONSTANT.TXT

In VB CDK, LibInit Does Not Allow Local Memory Allocation

Article Number: Q77836
MODIFIED: 7-NOV-1991

Summary:

LIBINIT.ASM provided with the Visual Basic Custom Control Development Kit (CDK) does not provide support for local memory allocation. This is because LIBINIT.ASM does not perform a **LocalInit**() Windows API call to initialize the local heap.

LibInit is an example of the simplest Windows entry routine for use with custom controls. You can either add the **LocalInit**() Windows API call to LIBINIT.ASM or use LIBENTRY.ASM provided with the Windows SDK package instead. Either of these methods will allow a custom control to perform local memory allocation. The changes to perform either of these methods is described below.

More Information:

To use LIBENTRY.ASM instead of LIBINIT.ASM with Visual Basic, you must modify the LibMain Declare statement provided in the CCInit.C routine:

The provided LibMain Declare statement has only three parameters. A fourth parameter for lpszCmdLine must be added.

```
BOOL FAR PASCAL LibMain (HANDLE hmod, HANDLE segDS, USHORT cbHeapSize, )
```

must be changed to:

```
BOOL FAR PASCAL LibMain (HANDLE hmod, HANDLE segDS, USHORT cbHeapSize, LPSTR
        lpszCmdLine)
```

If the LibMain Declare statement is not changed and LibEntry is used with a Visual Basic custom control, it will result in an "Unrecoverable Application Error" (UAE) message or other unpredictable results.

To modify LIBINIT.ASM to create local heap space, add the following lines of code to LIBINIT.ASM. Insert the lines after "push cx" and before "call LibMain":

```
push cx
        ; Add the following lines taken from LIBENTRY.ASM from Windows 3.0 SDK.
        ; if we have some heap then initialize it
        ; jump if no heap specified
jcxz      callc
        ; call the Windows function LocalInit() to set up the heap
        ; LocalInit((LPSTR)start, WORD cbHeap);
xor       ax,ax
cCall     LocalInit <ds, ax, cx>
         ; did it do it ok ?
or         ax,ax
         ; quit if it failed
jz         error
        ; invoke the C routine to do any special initialization
callc:
        ; End of added code.
call LibMain
```

After making the changes, LibInit needs to be reassembled with the Microsoft Macro Assembler.

No New Timer Events During Visual Basic Timer Event Processing

Article Number: Q78599
Modified: 13-DEC-1991

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 64767 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 it's message queue. If there is already 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.

References

Visual Basic Online Help Example Errors

Article Number: Q78772
Modified: 20-DEC-1991

Summary:

There are several code examples in Visual Basic's online Help that do not behave as expected if actually copied and run.

> Note: The corresponding examples in the "Visual Basic: Language Reference" manual contain the same errors.

Under the topic "ActiveControl, Active Form Properties," the second example demonstrating these properties contains an omission of the Clipboard object. When copied and run as is, the error message "Method Not Applicable For This Object" will be displayed.

The ActiveControl example contains the line in the EditCut_Click event procedure for the menu item:

```
SetText Screen.ActiveControl.SelText
```

This should be changed to read:

```
Clipboard.SetText Screen.ActiveControl.SelText
```

Under the topic titled "Fonts Property," the example shows setting the FontName = "". This will cause a run-time error "Invalid Property Value."

The example will also fail when attempting to select a printer FontName as the screen FontName where no associated screen font exists under Windows. For example, when the printer LinePrinter font is selected for the screen, an error will occur because the screen does not support this font. The examples for the topics "FontName Property" and "FontCount Property" if modified as suggested in the online Help to print the available printer fonts to the screen will fail for the same reason.

The example for the Fonts Property follows:

'Fonts Property Example

```
Sub Form_Click ()
    ' A static variable.
    Static X%
    ' Keep screen text.
    AutoRedraw = -1
    ' Check for last font.
    If X% = Printer.FontCount Then
        ' Set X%.
        X% = 0
    ' Print blank line.
        Print
    ' Reset to default font.
        FontName = ""
    End If
    ' Print header.
    If X% = 0 Then Print "Printer Fonts"
    ' Set FontName.
    FontName = Printer.Fonts(X%)
    ' Print message.
```

```
    Print X%; "This is " + FontName + " font"
      ' Set X%.
    X% = X% + 1
End Sub
```

As stated above, this fails in two ways. The line resetting the FontName property is syntactically incorrect. Also, the logic may fail because of no corresponding screen font for the printer font.

Modifying the example to address both problems requires an On Error trap routine and saving the values of FontName, FontBold, and FontSize explicitly. The following example works properly.

```
      'Fonts Property Example
Sub Form_Click ()
    On Error GoTo errHandler
    Static x%
    Static savename$, savebold%, savesize
      ' Keep screen text.
    AutoRedraw = -1
       ' Check for last font.
    If x% = Printer.Fontcount Then
      ' Set X%.
         x% = 0
       ' <<<add this! Reset to default font.
         FontName = savename$
         FontSize = savesize
         FontBold = savebold%
      ' Print blank line.
         Print
    End If
    If x% = 0 Then
       ' Print header.
         Print "Printer Fonts"
      ' save all these
         savename$ = FontName
         ' to original settings
         savebold% = FontBold
       ' now
         savesize = FontSize
    End If
      ' Set FontName.
    FontName = Printer.Fonts(x%)
    Print x%; "This is " + FontName + " font"    ' Print message.
ExitSub:
    x% = x% + 1 ' Set X%.
Exit Sub

errHandler:
    Print x%; "This is " + Printer.Fonts(x%) + " name"
    Resume ExitSub:
End Sub
```

Right Mouse Button Causes Remote Control Menus in Visual Basic

Article Number: Q78773
Modified: 20-DEC-1991

Summary:

The mouse behaves unexpectedly under the following conditions:

1. A Visual Basic program is run from the environment (VB.EXE) or an executable using the run-time module.

2. The program has a form that contains menus.

3. While holding a menu open with the left button, if you click the right mouse button, the mouse selection appears to be inactivated.

4. Moving up or down the menu while maintaining the left button in the down position causes no selection until you get several inches below the pop-up (or pull-down) menu.

5. At that point, the mouse causes selection again from the remote position.

This problem occurs when running a Visual Basic program that has menus. It requires a mouse with two buttons and has been reported with both Microsoft and Logitech mice.

Steps to Reproduce Problem

1. Run the Cardfile program that comes with Visual Basic in the Samples subdirectory.

2. Put the mouse cursor on one of the menu labels and press the left mouse button to activate it.

3. While continuing to hold down the left button, move the cursor to a menu item within the pop-up menu; this will highlight the menu item.

4. While holding the left button down, click the right button once. The menu item should no longer be highlighted.

5. Move the mouse from the item you were selecting. Observe that the mouse no longer activates submenus, and the menu does not retract.

6. Continue to move the mouse down from the menu. At some point, the highlighting of the submenu items will be activated again.

7. Upon stopping on a submenu item and releasing the left button, that menu command will execute.

   Note: This behavior also occurs if you open a menu and, while holding down the left button, you click the right button anywhere on the screen.

"Out of Memory" May Lead to Unstable Visual Basic Environment

Article Number: Q78774
Modified: 17-DEC-1991

Summary:

Once an "Out of Memory" error occurs in a session in the Visual Basic development environment (VB.EXE), subsequent operations may be affected. Repeated alert message boxes may appear during the session, primarily the "Out of Memory" error message, but other errors may occur in unexpected sequence.

Generally, the best solution is to exit the environment completely and then re-enter to a new session. Continued operations after the environment becomes unstable can lead to an "Unrecoverable Application Error" (UAE) message, and can possibly cause a system hang.

More Information:

Steps to Reproduce Problem

1. Load a text file into the global module (such as the WINAPI.TXT file that accompanies the Windows Programmer's Guide and Online Resource) that exceeds the 64 K data segment size of the global module.

2. From the Code menu, choose Load Text. In the subsequent dialog box, select an oversize text file and press ENTER.

3. When the "Out of Memory" message is displayed, choose OK in the alert box.

4. From the File menu, choose New Form or New Module.

5. Another "Out of Memory" message is displayed, followed by a another message that Visual Basic is "Out of Memory."

6. Choose OK and attempt to open a file.

Another "Out of Memory" error will occur, and the file will not load completely. The environment is corrupted and in an unstable state. At this point, the only solution is to exit Visual Basic.

References

Multiline Text Box Contents Not Grayed When Control Disabled

Article Number: Q78892
Modified: 18-DEC-1991

Summary:

When the MultiLine property of a text box is True (-1) and the Enabled property is False (0), text inside the text box is incorrectly displayed black, rather than gray.

More Information:

To work around this problem, set the ForeColor property of the text box to gray when the Enabled property is set to False (-1) as shown in the example below.

```
        '####In the global module:
         'from CONSTANT.TXT
  Global Const WINDOW_TEXT = &H80000008
  Global Const GRAY_TEXT = &H80000011

        '####In the form:
        'to disable a multiline text box
  Text1.Enabled = 0
  Text1.ForeColor = GRAY_TEXT          'gray

        'to enable a multiline text box
  Text1.Enabled = -1
  Text1.ForeColor = WINDOW_TEXT     ' black
```

Incorrect Jumps in Visual Basic Online Help "How To" Section

Article Number: Q78895
Modified: 18-DEC-1991

Summary:

In the Visual Basic online Help, under the How To option, under the Graphics topic, there are two incorrect jumps. If you select the Displaying Graphics at Run Time or Displaying Graphics on a Form topic, online Help will jump to an incorrect Help screen.

More Information:

Workaround

For online Help with either of these topics, do the following:

1. From the Help menu, choose Index.

2. Choose the Search button.

3. Type the word "graphics," or scroll down the list to the graphics topic.

4. Choose the Show Topics button.

5. The Displaying Graphics on a Form and Displaying Graphics at Run Time topics will both appear in the Go To list box. If you select either of these topics, Help will display the correct screen.

Steps to Reproduce Problem

1. From the Visual Basic (VB.EXE) Help menu, choose Index.

2. Choose the How To option.

3. Scroll down to the Graphics topic.

4. Select the Displaying Graphics at Run Time topic or the Displaying Graphics on a Form topic.

When selected, both of these topics jump to the wrong Help screens.

Selecting the Displaying Graphics At Run Time option will bring you to the Adding and Deleting Menu Commands At Runtime screen. Selecting the Displaying Graphics On A Form option will take you to the Displaying Graphics At Run Time screen.

VB SETUP.EXE "Insufficient Disk Space on: C:\WINDOWS"

Article Number: Q78961
Modified: 18-DEC-1991

Summary:

Visual Basic will display the message

> Error - Insufficient disk space on: C:\WINDOWS

during setup if there is less than 330K of space available to Windows on the drive where Windows resides, which may be different than the drive on which you are installing Visual Basic.

More Information:

At the beginning, SETUP.EXE for Visual Basic copies the files VBSETUP.EXE and VBRUN100.DLL (approximately 55K and 272K bytes, respectively) into the Windows subdirectory. If there is not enough space on the drive where Windows resides (such as in C:\WINDOWS), Visual Basic will display the "Error - Insufficient disk space on: C:\WINDOWS" message.

This is the disk space available to Windows just before setup. This may differ from the amount of space reported at the MS-DOS command prompt outside of Windows because of temporary files that Windows creates during operation.

VBSETUP.EXE is deleted when setup is completed. VBRUN100.DLL is copied over to the Visual Basic subdirectory, but is not deleted from the Windows subdirectory.

Creating Nested Control Arrays in Visual Basic

Article Number: Q79029
Modified: 20-DEC-1991

Summary:

Creating an array of picture controls or frames with an array of child controls (such as command buttons) within each element of the parent array is not possible in Visual Basic without making use of the Windows API functions **SetParent**() and **GetFocus**().

Because Visual Basic does not support overlapping controls, creating controls and then simply moving them into position within previously created controls does not function correctly. Using the Windows API functions in conjunction with the Load and Unload methods can circumvent this problem and allow dynamic, flexible structures to be created during execution.

More Information:

This article explains how to get the following control structure

Parent Control	Child Controls			
Picture1(1)	Command1(1),	Command1(2),	Command1(3),	Command1(4)
Picture1(2)	Command1(5),	Command1(6),	Command1(7),	Command1(8)
Picture1(3)	Command1(9),	Command1(10),	Command1(11),	Command1(12)
Picture1(4)	Command1(13),	Command1(14),	Command1(15),	Command1(16)

(where Picture1 and Command1 are control arrays).

The following example uses the two API functions **GetFocus**() and **SetParent**() to establish the proper parent/child relationships between an array of parents (such as picture controls) and an array of children (such as command buttons) where each child array is within one element of the parent control.

The function **GetFocus**() requires no parameters. The function **SetParent**() requires two:

Parameter	Type/Description
hWndChild	HWND Identifies the child window
hWndNewParent	HWND Identifies the new parent window

The return value identifies the previous parent window.

This example also demonstrates how Windows handles a drag and drop if parentage was set at run time. If the control is dragged to another control of the same type as the previous parent; when released, the control assumes the same relative position it had in the previous parent. The program demonstrates that before unloading controls nested using **SetParent**() and moved during the run, parentage should be returned to the original hierarchy. This is to avoid the possibility of "Unrecoverable Application Error" (UAE) messages that could occur due to conflicting messages to Windows.

Create the following controls:

Control	CtlName	Property	Setting
Form	Form1		
Picture	Picture1()	Index=1	
Command button	Command1()	Index=1	

Command button	Loadall	Taborder=0,	Caption="Load All"
Command button	Loadsome	Taborder=1,	Caption="Load Four"
Command button	Changemode	Taborder=2,	Caption="DragMode=0"
Command button	Unloadall	Taborder=3,	Caption="Unload All"

Note: The placement of the original picture box and command button is important. The picture should be created first and the command button drawn WITHIN the picture box. The placement of the other controls at design time is not critical. They are resized and moved at run time.

Code	Purpose
Loadall_Click	Loads all parent controls and all child controls
Loadsome_Click	Loads all parent controls and four child controls
Unloadall _Click	Unloads all controls. Must reset parentage to the original first!
Changemode_Click	Changes DragMode between auto and manual modes
Resetparent	General procedure to reset parentage for Unload and Close
Cplace	General procedure to place the four command buttons
Form_Resize	Code to maintain original size
Form_Load	Sets initial size and properties of controls
Form_Unload	Calls Unloadall_Click to avoid conflicting Windows messages
Picture1_DragDrop	Handles setting of parentage to user actions
Command1_Click	Sets captions to reflect change in position and state

Run the example and try all the buttons. Toggle the DragMode on and off and drag the command buttons from one picture to another. Add the following code to your program:

Global Module

```
Declare Function Setparent% Lib "user" (ByVal h%, ByVal h%)
Declare Function GetFocus% Lib "user" ()
Global Handle2Child As Integer
Global Handle2Parent As Integer
Global dragstate, loadstate, toggle, innernum As Integer
Global I, N, K As Integer
Global xoffset, yoffset, cmdnum, childsize, parentsize As Integer
Global Const maxouter = 4
Global Const maxinner = 4
Option Base 1
      'array=maxinner*maxouter
Global storecaption(16) As String
```

Form1

```
Sub Picture1_DragDrop (index As Integer, Source As Control, X As Single, Y As Single)
      'Procedure for control array of parent Picture Boxes
    Picture1(index).SetFocus
    Handle2Parent = GetFocus()
    Source.SetFocus
    Handle2Child = GetFocus()
    ret% = SetParent(Handle2Child, Handle2Parent)
    Source.caption = Mid$(Source.caption, 1, 1) + "/" + LTrim$(RTrim$(Str$(index)))
End Sub
```

```
Sub Form_Load ()
    Form1.width = Screen.Width - Screen.Width \ 8
    Form1.Height = Screen.Height \ 2
```

```
    Form1.Backcolor = &HFFFF00
    Form1.Caption = "Nested Control Arrays"
    Picture1(1).Visible = 0
    Command1(1).Visible = 0
    parentsize = CInt((Form1.Width \ maxouter) * .8)
    childsize = CInt((2 * parentsize \ maxinner) * .6)
    Picture1(1).Height = parentsize
    Picture1(1).Width = parentsize
    Command1(1).Height = childsize
    Command1(1).Width = childsize
    cplace loadall, 1
    cplace loadsome, 2
    cplace Changemode, 3
    cplace Unloadall, 4
End Sub
```

```
Sub Resetparent ()
      'Function to clean up parentage before unload
    Picture1(1).SetFocus
    Handle2Parent = GetFocus()
    For I = innernum To 1 Step -1
        Command1(I).SetFocus
        Handle2Child = GetFocus()
        ret% = SetParent(Handle2Child, Handle2Parent)
    Next I
End Sub
```

```
Sub Command1_Click (index As Integer)
      'Procedure for control array of Buttons
    If toggle Then
        Command1(index).Caption = storecaption(index)
        toggle = 0
    Else
      ' change caption to reflect state
        storecaption(index) = Command1(index).Caption
        Command1(index).Caption = "ON"
        toggle = -1
    End If
End Sub
```

```
Sub Form_Unload (Cancel As Integer)
      'Cleans up before program exit Unloadall_Click
End Sub
```

```
Sub Changemode_Click ()
      'Toggles between automatic & manual dragmodes
    If loadstate Then
        If Not dragstate Then
            For I = 1 To innernum
      'automatic
                Command1(I).DragMode = 1
      'reset flag
                dragstate = -1
            Next I
            changemode.Caption = "DragMode=1"
```

```
        Else
            For I = 1 To innernum
      'manual
                Command1(I).DragMode = 0
      'reset flag
                dragstate = 0
            Next I
            changemode.Caption = "DragMode=0"
        End If
    End If
End Sub
```

```
Sub Unloadall_Click () 'Unloads all dynamically created controls ONLY
    Select Case loadstate
    Case 1
      'Must call prior to unload to avoid UAEs
        Resetparent
        For I = maxouter To 1 Step -1
            For N = maxinner To 1 Step -1
                cmdnum = ((I - 1) * 4) + N
                If cmdnum <> 1 Then Unload Command1(cmdnum)
            Next N
            If I <> 1 Then Unload Picture1(I)
        Next I
      'Can't unload controls
        Command1(1).Visible = 0
      'created at design time so hide!
        Picture1(1).Visible = 0
      'reset flag for load routines
        loadstate = 0
        changemode.Enabled = 0
    Case 2
      'Must call prior to unload to avoid UAEs
        Resetparent
        For I = maxouter To 1 Step -1
            If I = 1 Then
                For N = maxinner To 2 Step -1
                    Unload Command1(N)
                Next N
            End If
            If I <> 1 Then Unload Picture1(I)
        Next I
      'Can't unload controls
        Command1(1).Visible = 0
      'created at design time so hide!
        Picture1(1).Visible = 0
      'reset flag for load routines
        loadstate = 0
        changemode.enabled = 0
    End Select
End Sub

    'Loads all parents and one set of children
```

```
Sub Loadsome_Click ()
    ' to demonstrate drag and drop
```

```
    If loadstate = 0 Then
        changemode.Enabled = -1
        Command1(1).Move 0, 0
        For I = 1 To maxouter
            If I <> 1 Then
      ' can't load control created at design time
                Load Picture1(I)
            End If
            Picture1(I).Move -Picture1(1).Width, -Picture1(1).Height, parentsize, parentsize
      'load off-screen /\
            Picture1(I).Visible = -1
            Picture1(I).SetFocus
      'get handle by API call
            Handle2Parent = GetFocus()
            If I = 1 Then
                For N = 1 To 4
                    If N <> 1 Then
      'can't load control created at design time
                      Load Command1(N)
                    End If
                    xoffset = Picture1(I).ScaleWidth \ 4 -
                    Command1(N).width \ 2 +((N - 1) Mod 2) * (Picture1(I).ScaleWidth \ 2)
                    If N > 2 Then
                        yoffset = Picture1(I).ScaleHeight \ 2 + (Picture1(I).ScaleHeight \ 4 -
                            Command1(N).Height \ 2)
                    Else yoffset =(Picture1(I).Scaleheight\4-Command1(N).Height\2)
                    End If
                    Command1(N).Move xoffset, yoffset
                    Command1(N).Visible = -1
                    Command1(N).SetFocus
      'get handle by API call
                    Handle2Child = GetFocus()
      'Call API function
                    ret% = SetParent(Handle2Child, Handle2Parent)
                    Command1(N).Caption = LTrim$(RTrim$(Str$(N))) + "/" + LTrim$(RTrim$(Str$
                        (I)))
                Next N
            End If
        Next I
        xoffset = ((Form1.Scalewidth \ maxouter) - Picture1(1).Width) \ 2
        Picture1(1).Move xoffset, 0
        For I = 2 To maxouter
            Picture1(I).Move (I - 1) * (Form1.Scalewidth \ maxouter) + xoffset, Picture1(I - 1).Top
        Next I
      'set global loop maximum
        innernum = 4
        loadstate = 2
    End If
End Sub

      'Loads all parents and children in nested structure
-------------------------------------------------------------------------------
Sub Loadall_Click ()
    If loadstate = 0 Then
        changemode.Enabled = -1
        Command1(1).Move 0, 0
      ' size command button proportionally
```

```
        For I = 1 To maxouter
    'can't load control created at
            If I <> 1 Then Load Picture1(I)
            'design time load off-screen:
            Picture1(I).Move -Picture1(1).Width, -Picture1(1).Height
            Picture1(I).Visible = -1
            Picture1(I).SetFocus
    'get handle by API call
            Handle2Parent = GetFocus()
            For N = 1 To maxinner
                cmdnum = ((I - 1) * 4) + N
                If cmdnum <> 1 Then
    'can't load control created at
                    Load Command1(cmdnum)
    'design time
                End If
                xoffset =((N-1) Mod 2)*(Picture1(I).ScaleWidth(maxinner\2))
                If N > 2 Then
                   yoffset = Picture1(I).ScaleHeight \ 2
                Else yoffset = Picture1(I).ScaleTop
                End If
                Command1(cmdnum).Move Picture1(I).ScaleWidth \ 8 + xoffset,
                        Picture1(I).ScaleHeight \ 8 + yoffset
                Command1(cmdnum).Visible = -1
                Command1(cmdnum).SetFocus
    'get handle by API call
                Handle2Child = GetFocus()
    'Call API function
                ret% = SetParent(Handle2Child, Handle2Parent)
            Next N
    'Caption the control array buttons
        Next I
        For K = 1 To (maxinner * maxouter)
            Command1(K).Caption = LTrim$(RTrim$(Str$(K)))
        Next K
        xoffset = ((Form1.Scalewidth \ maxouter) - Picture1(1).Width) \ 2
        Picture1(1).Move xoffset, 0
        For I = 2 To maxouter
            Picture1(I).Move (I - 1) * (Form1.Scalewidth \ maxouter) + xoffset, Picture1(I - 1).Top
        Next I
    'set global loop maximum
        innernum = 16
        loadstate = 1
    End If
End Sub
```

```
Sub Form_Resize ()
    Form1.Width = Screen.Width - Screen.Width \ 8
    Form1.Height = Screen.Height \ 2
End Sub

    'size static controls
```

```
Sub cplace (dummy As Control, num As Integer)
    theheight% = parentsize + childsize * 2
    dummy.Move (Form1.width \ 4) * (num - 1) + parentsize \ 10, theheight%, parentsize, childsize
```

```
End Sub
```

References

Visual Basic List Box Won't Open If Resized at Run Time

Article Number: Q79030
Modified: 19-DEC-1991

Summary:

When you click the down arrow of the drive list box control, the drive list box will not open if it has been resized at run time. The Width property is Read/Write at run time. However, if it is changed at run time, the drive list box won't open. This is true even if it is restored to its original value before attempting to open it.

Note also that page 11 of the "Microsoft Visual Basic: Language Reference" incorrectly lists the Height property of the drive list box as being Read/Write at run time, but Height is actually Read-Only at run time.

Note: Neither the directory list box nor the file list box is affected by run-time resizing.

Steps to Reproduce Problem

1. Click the drive list box control icon on the Toolbox. Draw a drive list box on the form. Resize the drive list box to any size you desire. At run time, the drive list box will correctly open when you click the down arrow.

2. Create three command buttons on the form. Caption them "Narrow," "Wider," and "Restore."

3. Insert the following code:

   Note: The example assumes a starting dimension of 2055 wide (user alterable) by 315 high (the standard height in twips).

```
Sub Command1_Click ()
    drive1.WIDTH = 1025             ' Narrow
End Sub
```

```
Sub Command2_Click ()
    drive1.WIDTH = 4110             'Wider
End Sub
```

```
Sub Command3_Click ()
    drive1.WIDTH = 2055             'Restore
End Sub
```

4. Run the example.

5. Open the drive list box. Click on Narrow or Wider.

6. Try to open the drive list box again. It fails to open.

7. Click Restore. Again try to open the drive list box. It fails to open.

List of Visual Basic Third-Party Add-On Products

Article Number: Q78963
Modified: 30-JAN-1992

Summary:

This article contains the following sections:

Please send any additions or corrections to this list to:

Michael Risse
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
Tel. (206) 882-8080
Fax (206) 93-MS-FAX (206-936-7329)

Custom Controls and DLLs

ADDE
17, Rue Louise Michel
92301 Levallois-Perret, France
Contact: Xavier Ledur +33-1-47-58-78-41

Map Custom Control

The Map Custom Control is a window in which one or more overlapped geographical maps (cities, countries, networks) can be displayed. Zooming and positioning functions will be integrated in the control. The custom control will be able to load maps from "Cartes et Bases Windows," an existing Microsoft+ Windows+ graphical environment map package. Sample maps from ADDE catalog will be integrated with this control. Available fourth quarter.

Autodesk, Inc.
2320 Marinship Way
Sausalito, CA 94965

Contact: (415) 332-2344

Autodesk Animation Player for Visual Basic This custom control permits Visual Basic users to easily add animation to their applications. The control plays industry-standard FLI and FLC animations at remarkable speed from hard disks or CD-ROMs.

Coromandel
70-15 Austin Street, Third Floor
Forest Hills, NY 11375
Contact: (800) 535-3267, (718) 793-7963
Fax (718) 973-9710

DbControls

Database custom controls for Visual Basic. Build database applications without writing any code. Uses ObjecTrieve's database engine, with support for binary large objects (BLOBS), multiple variable-length fields in the same record, unlimited number of indexes, and non-contiguous multi-key parts.

DbControls for dBASE

Database custom controls for Visual Basic. Read and write dBASE III files without writing any code. Create new dBASE files from your Visual Basic Applications.

DbControls for Btrieve

Database custom controls for Visual Basic. Read and write Btrieve files without writing any code.

Crescent Software, Inc.
32 Seventy Acres
West Redding, CT 06890
Contact: Ethan Winer (203) 438-5300

QuickPak Professional for Windows

QuickPak Professional for Windows contains custom controls and a general purpose set of utilities for use with Microsoft Visual Basic programming system. QuickPak Professional for Windows provides routines for quickly sorting and searching data, performing fast file operations, expression evaluation, and other useful tasks. (Available Fall, 1991)

Desaware
5 Town & Country Village #790

San Jose, CA 95128
Contact: Gabriel Appleman (213) 943-3305

Custom Control Factory
An interactive development tool for creating custom controls including Animated Pushbuttons, Multistate Buttons, enhanced buttons, checkbox and option button controls for the Visual Basic system.

FutureSoft
1001 South Dairy Ashford, Suite 203
Houston, TX 77077
Contact: Teri Taylor (713) 496-9400

DynaComm
Full line of communication from Asynchronous DEC Connectivity to IBM 3270 Connectivity. By midsummer, with each DynaComm product, Visual Basic system users will be able to visually link their applications to DynaComm using DynaComm Custom Controls.

INSYS
268 Rue du Faubourg Saint-Antoine
75012 Paris, France
Contact: M. Quentin +33-1-40-04-6-36

Insys Classes
Insys Classes is a collection of Visual Basic programming system Custom Controls for business oriented, computing and communications applications, including: structured text fields (numeric, alphanumeric, masked input), hierarchical list boxes, structured list boxes, date/time management controls with spin buttons, CPIC control, and a simple spreadsheet control. Available in September 1991 in French and English versions.

MicroHelp, Inc.
4636 Huntridge Drive
Roswell, GA 30075
Contact: Mark Novisoff (404) 594-1185

VB Tools
Designed to add "pizazz" to Visual Basic programs, VB Tools includes numerous custom controls, graphics special effects, how to use Windows API services, utility modules, a program providing $INCLUDE capabilities, and more.

OutRider Systems
P.O. Box 271669
Houston, TX 77277-1669
Contact: Jim Nech (713) 521-0486

ButtonTool
With ButtonTool, developers can easily create many new button types and styles using bitmaps, icons and metafiles as backgrounds.

Edit Tool
Mask for custom edit box that formats dates, times, dollars, and numbers.

Pinnacle Publishing
P.O. Box 8099
Federal Way, WA 98003
Contact: David Johnson (800) 231-1293 or (206) 941-2300

Graphics Server for Visual Basic

Graphics Server for Visual Basic is a custom control for integrating graphing and charting capabilities into Visual Basic applications. Include pie charts, bar charts and a variety of other graphs in 2D or 3D.

Prescription Software Inc.
4857 Havana Drive
Pittsburg, PA 15239
Contact: Jeff O'Donnell (412) 798-9454

Drover's Professional Toolbox for Visual Basic

A dynamic link library of 200 functions and several custom controls. Provides many formatted edit controls and a full featured spreadsheet for use in Visual Basic. Available October, 1991.

Scientific Software Tools, Inc.
30 East Swedesford Road
Malvern, PA 19355
Contact: Elise Furman (215) 889-1354, Fax (215) 889-1630

DriverLINX/VB

DriverLINX/VB is a custom control for Visual Basic that adds a high- performance data-acquisition engine to the Visual Basic Toolbox. Quickly create sophisticated virtual instruments that you could only dream of under DOS, in just days, using DriverLINX/VB and a data- acquisition board. DriverLINX/VB provides a standardized, abstract (hardware-independent) Interface to over 100 services for creating foreground and background tasks to perform analog input and output, digital input and output, time and frequent measurement, event counting, pulse output and period measurement. In addition to basic I/O support, the DriverLINX/VB engine also provides sophisticated built-in capabilities to handle memory and data buffer management, a rich selection of starting and stopping trigger events, e.g., pre-mid- point and post-triggering protocols, and extensive error-checking and reporting capabilities.

Sheridan Software Systems, Inc.
65 Maxess Road
Melville, NY 11747
Contact: Joseph Modica (516) 753-0985, fax (516) 293-4155

VB Extenders

3-D Widgets is a collection of 6 custom controls which support 3 dimensional text boxes and controls on Visual Basic forms (currently shipping). Additional extensions are planned for release in Fall, 1991.

Creating .DLLs:

Programmers are able to create .DLL files that are callable from Visual Basic using any of the following language tools. There are only two qualifications: the .DLL must use Pascal calling conventions (the standard for Microsoft Windows) and the .DLL can not employ callbacks to the Visual Basic executable.

- Microsoft C 6.0 with the Microsoft Windows Software Development Kit
- Microsoft Macro Assembler 6.0
- Microsoft FORTRAN 5.1
- Microsoft COBOL 4.5
- Microsoft Quick C for Windows
- Borland Turbo Pascal for Windows
- Borland C++
- Watcom C

- Zortech C++

- Data Access/Connectivity

Aaerdeus, Inc.
302 College Avenue
Palo Alto, CA 94306
Contact: Randy Burns (415) 325-7529

SQL Express

SQL Express is a dynamic link library and set of example programs that allow the Microsoft SQL Server to be used with Microsoft Visual Basic programming system.

Apex Software Corporation
4516 Henry Street, Suite 401
Pittsburg, PA 15213
Tel. (412) 681-4343
Contact: Richard F. DiGiovani (818) 594-7293

Agility/VB

Agility/VB is a database developer's tool for Visual Basic based upon Apex's powerful Apex Database Library. It is provided as a set of DLL functions callable from Visual Basic programs, which the programmer defines and relates to eachother using a graphical View Editor. Provides complete access to dBASE IV compatible files.

Attachmate
13231 S.E. 36th Street
Bellevue, WA 98006
Contact: Posy Gering or Mike New (800) 426-6283

Extra for Windows 3.2

Extra for Windows 3.2 gives Visual Basic developers access to IBM mainframes. Programs can be written to automatically integrate mainframe information with PC applications using DDE, DLL calls, and Visual Basic custom controls.

Blue Rose Software
Box 29574 Atlanta, GA 30359-0574
Contact: Richard Denton (404) 717-1220

DATABasic

A B-tree database engine for use with Visual Basic featuring speed, flexibility, small libraries, ease of maintenance, and rapid software development. It provides an integrated devlopment environment. DATABasic eliminates an entire class of programming bugs -- synchronization bugs between code and databases.

Borland International
1800 Grenn Hill Rd
Scotts Valley, CA 95067
Contact: 408-439-1639

Paradox Engine Version 2.0

Includes DLL for developing Windows applications. You can create, read, and write Paradox tables, records, and fields. Supports multiuser database functions such as multiuser file locking, record locking, and password protection. Applications created with the Paradox engine ship run time and royalty free.

Channel Computing, Inc.

53 Main Street
Newmarket, NH 03857
Contact: Max Klein (603) 659-2832

Forest & Trees

Forest & Trees is a new Data Access and Reporting Tool that lets Visual Basic system users build an "electronic dashboard" to collect, combine, and automatically monitor information from a wide range of spreadsheets, database files and database servers.

CNA Computer Systems Engineering, Inc.
P.O. Box 70248
Bellevue, WA 98007
Contact: John Evans (206) 861-4736

ConnX

ConnX is a connectivity tool allowing record level communication between Visual Basic applications and indexed or sequential VAX RMS files while supporting user and file level security.

Comsoft
P.O. Box 3835
Postal Station "D"
Edmonton, AB T5L 4K1
Canada
Contact: (403) 489-5994 Fax: (403) 486-4335

vxBase

DLL that allows Visual Basic programmers to create xBASE applications for Windows in hours. It's all the functions--vxAppendBlank through vxZap (86 functions in all). Browse object supports user-defineable tasbles, on-screen editiong, and visual relationships. Available as shareware on MSBASIC forum on CompuServe or directly from Comsoft.

Copia International
1342 Avalon Court
Wheaton, MD 60187
Contact: Dorothy Gaden (708) 682-8898

AccSys for Paradox

AccSys for Paradox with the Microsoft Visual Basic programming system for Windows provides the programmer total control over Paradox table files, primary and secondary index files.

Coromandel
70-15 Austin Street, Third Floor
Forest Hills, NY 11375
Contact: (800) 535-3267, (718) 793-7963
Fax (718) 973-9710

ObjecTrieve for Visual Basic

ObjecTrieve/VB is an ISAM DLL for Microsoft Windows and Visual Basic. ObjecTrieve/VB is capable of storing and retrieving binary large objects (BLOBS)--such as scanned images, video, documents, bitmaps, etc. Includes Visual Basic declarations and sample code.

DbControls

Database custom controls for Visual Basic. Build database applications without writing any code. Uses ObjecTrieve's database engine, with support for binary large objects (BLOBS), multiple variable-length fields in the same record, unlimited number of indexes, and non-contiguous multi-key parts.

DbControls for dBASE

Database custom controls for Visual Basic. Read and write dBASE III files without writing any code. Create new dBASE files from your Visual Basic Applications.

DbControls for Btrieve

Database custom controls for Visual Basic. Read and write Btrieve files without writing any code.

Integra SQL

Integra SQL complements and extends the Visual Basic system by providing high-performance relational database functionality, including building, querying, updating and reporting of facilities.

DatTel Communications Systems, Inc.
3508 Market Street, Suite 415
Philadelphia, PA 19104
Contact: Ravi Gururaj (215) 564-5577

DataLIB

Dynmanic link library (DLL) that allows Visual Basic programmers to read and write Excel. Lotus 1-2-3, dBASE, and DIF, SYLK and ASCII files. Includes all Visual Basic declarations and sample application.

Daytris Inc.
81 Bright Street, Suite 1E
Jersey City, NJ 07302
Contact: Todd C. Fearn (201) 200-0018

CDB for Windows

A sophisticated database toolkit for Windows developers offering multi-user ISAM functionality, relational and network data models, client server implementations, portability to MS-DOS and UNIX platforms, and royalty free distribution of object files.

Digital Communications Associates, Inc.
1000 Alderman Drive
Alpharetta, GA 30202-4199
Contact: Margaret Owens (404) 442-4521

IRMA Workstation for Windows and CROSSTALK for Windows

IRMA Workstation for Windows' (IWW) Standard IRMA Scripting Language and the Crosstalk products' Crosstalk Application Scripting Language (CASL) enable developers to write scripts that transfer information to and from mainframes or information services using Microsoft Visual Basic applications through dynamic data exchange (DDE). Supports XModem and ZModem transfer protocols.

Distinct Corporation
P.O. Box 3410
Saratoga, CA 95070-1410
Contact: Chris Apap-Bologna (408) 741-0781

Distinct TCP/IP Software Development Kit

Berkeley Sockets, RPC/XDR and NFS toolkit for the Microsoft Windows environment -- includes Visual Basic declarations. Allows developers to write custom TCP/IP network applications or distributed applications for Windows. Accessed via DLL.

Dome Software Corporation
655 West Carmel Drive, Suite 151
Carmel, IN 46032

Fax: 317-573-8109
Contact: Ken Jones (317) 573-8100

Parley

Parley is a client server product which provides access to VAX or mainframe based data. It provides a network independent communication layer which fully integrates a Visual Basic application into a variety of corporate data sources (SQL and non-SQL sources).

ETN Corporation
RD4 Box 659
Montoursville, PA 17754-9433
Contact: Wynne Yoder (717) 435-2202

PowerLibW

PowerLibW is a library (DLL) of over 90 functions and a DBMS server that provides dBASE III and IV and Clipper compatible I/O which the Microsoft Visual Basic programmer may access. Supports expressions, filters, indexes, memos, relations, and multiple database access.

PowerShoW

PowershoW is a dynamic link library (DLL) which manages BMP, TARGA, and TIFF images for use in Windows.

The Frustum Group, Inc.
122 East 42nd Street, Suite 1700
New York, NY 10168
Contact: Chris Davis (212) 984-0760 or (800) 548-5660
Fax: (212) 687-8119

TransPortal PRO

A data-exchange toolkit that integrates Visual Basic applications with on-line host applications (3270, 5250, or VT100). DLL can be used to read from, write to, and send keystrokes directly to host application. Includes Visual Basic declarations.

FutureSoft
1001 South Dairy Ashford, Suite 203
Houston, TX 77077
Contact: Teri Taylor (713) 496-9400

DynaComm

By midsummer, with each DynaComm product, Visual Basic system users will be able to visually link their applications to DynaComm using DynaComm custom controls. Planned to support IBM, HP, NEC, and Data General mainframes.

Groupe Bull
7, Rue Ampere
91343 Massy, France
Phone: +33-1-69-93-90-90

Affinity-Visual

Affinity-Visual will fully integrate the Microsoft Visual Basic system with Bull's Affinity product. Affinity-Visual will provide full Windows environment 3.0 graphical display services to existing host applications throughout Bull environments.

Gupta Technologies
1040 Marsh Road
Menlo Park, CA 94025

Contact: (415) 321-9500

SQLBase Server:
This is a multi-user SQL database engine which supports crash recovery, password protection, on-line backup, and remote monitoring. Gupta has DLLs which provide access to the server from Visual Basic client applications.

MDBS
PO BOX 6089
Lafayette, IN 47903
Fax: (317)448 6428
Contact: Gary Rush (317) 447-1122

MDBS VI
An ISAM that works with Visual Basic.

Microcom Inc.
55 Federal Road
Danbury, CT 06810
Contact: (800) 822-8224 or Howard Luxenberg (203) 730-4378

MicroCourier
A complete communication package for Windows for under $100. Includes sample applications written in Visual Basic, with full source code.

Microsoft Corporation
One Microsoft Way
Redmond, WA 98027
Contact: Microsoft Inside Sales (800) 227-4679

Microsoft Visual Basic Library for SQL Server
Write Visual Basic applications for Microsoft SQL Server using the Visual Basic Library for SQL Server.

Microsoft LAN Manager Toolkit for Visual Basic
The tools you need to customize your LAN Manager-based network using Microsoft Visual Basic. Includes a graphing facility for displaying performance information and other system statistics. Sample utilities for common network management and diagnostic applications.

NetManage, Inc.
20823 Stevens Creek Blvd., Suite 100
Cupertino, CA 95014
Contact: Sales Dept. (408) 973-7171 Dan Geisler

Chameleon TCP/IP for Windows
TCP/IP application package for Windows. Includes TELNET, FTP, TFTP, SMTP/mail, name services, PING, network management and diagnostics. Implemented as a Windows DLL callable from Visual Basic applications as both client and server.

RPC-SDK: ONC Development Tools
Comprehensive software development kit for building distributed applications under Windows 3.0 using Sun ONC RPC/XDR. Windows DLL callable from Visual Basic applications as RPC client and server.

NEWT/SDK
Comprehensive software development kit for Windows 3.0 TCP/IP communications protocol. Offers

the Visual Basic programmer direct access to the Berkeley 4.3BSD socket interface, FTP and SMTP.

Novell, Inc.
5918 West Courtyard Drive
Austin, TX 78732
Contact: Mary K. Ellsworth (512) 794-1488

Btrieve for Windows Developer's Kit

The Btrieve Developer's Kit is a complete toolkit that allows Visual Basic developers to write applications with Btrieve, Novell's key- indexed record manager.

Pioneer Software
5540 Centerview Drive, Suite 324
Raleigh, North Carolina 27608
Contact: Richard Holcomb (919) 859-2220. Sales: (800) 876-3101

Q+E Database Library

Q+E Database Library allows Visual Basic applications to access data stored in most major PC, LAN and mainframe systems.

Quadbase Systems, Inc.
790 Lucerne Drive, Suite 51
Sunnyvale, CA 94086
Fax: (408) 738-6980
Contact: Fred Luk (408) 738-6989

Quadbase-SQL for Windows

Quadbase-SQL for Windows, a DLL (dynamic link library), is a full- featured, compact and high performance relational database engine for Visual Basic programmers to build single and/or multi-user applications that require advanced database features and industry standard SQL. The system can directly access dBase IV, Lotus 123, Foxpro index, and Clipper index files.

Raima Corporation
3245 146th Place S.E., Suite 230
Bellevue, WA 98007
Contact: (206) 747-5570
Marketing contact: Bill Pieser

db_VISTA III Database Management System

The db_VISTA III Database Management System combines both relational and network model database technologies for high-performance Visual Basic application development. API can be easily called from Visual Basic for database application development. Sample application in Visual Basic available upon request.

SQLSoft
10635 N.E. 38th Place, Bldg. 24, Suite B
Kirkland, WA 98942
Contact: James O'Farrell (206) 822-1287

SQL SoftLink

Dynamic data exchange (DDE) connectivity to Microsoft SQL Server.

Rochester Software Connection
4909 Highway 52 North
Rochester, MN 55901

Contact: (507) 288-5922, (800) 829-3555

ShowCase WindowLink

ShowCase WindowLink allows you to link your Visual Basic applications to IBM's AS/400 midrange systems. A DLL with Visual Basic declarations and sample code.

Sequiter Software Inc.
#209, 9644-54 Ave.
Edmonton, AB, Canada T6E 5V1
To order: Tel. (403) 437-2410, Fax (403) 436-2999,
Europe Tel. +33.20.24.20.14, Europe fax +33.20.24.20.90

CodeBase 4.5

A complete multi-user library for database management. Compatible with dBASE IV/III, Clipper, and FoxPro 2.0 data, index and memo files. Includes a Windows DLL for Visual Basic and on-line documentation with Visual Basic declarations and examples.

Software Source
42808 Christy St. Ste 222
Fremont, CA 94538
Fax (415) 651-6039
Contact: Sam Cohen (415) 623-7854

VB/ISAM

VB/ISAM extends Visual Basic with a set of simple functions to read and write data file records by alphanumeric key. Capabilities include field-structured (Get and Put) or unstructured access, read next/previous/approximate record, variable-length records and keys, and very large records (up to 32KB) and files (up to 512MB).

TechGnosis, Inc.
One Park Place
621 N.W. 53rd Street, Suite 340
Boca Raton, FL 33487
Contact: Keith Toleman (407) 997-6687

SequeLink

Client-server data access for Visual Basic system. Provides access to OS/2, UNIX, VAX/VMS, and AS400 servers. Supported databases include Oracle, Sybase, Ingres, SQL Server, DBM, RDB, SQL 400.

SequeLink Engine

A software development toolkit enabling workstation access to host- based data and applications. Extends the functionality of the company's SequeLink client/server architecture by enabling host operating systems, applications, and non-relational DBMSs to act as servers for Windows applications.

Unelko Corporation
7428 E. Caren Drive
Scottsdale, AZ 85260
Contact: Tony Pitman (602) 991-7272

Bridgit

A Dynamic Link Library that contains functions to allow full access to dBase III files, indexes, and memos. Two versions will be available: one for dBase III and the other for Clipper index files. Shipping January, 1992.

Wall Data Incorporated
17769 N.E. 78th Place
Redmond, WA 98052
Contact: Catherine Rudolph (Marketing Communications) (206) 883-4777
Fax: (206) 885-9250

Rumba Application Development Kit

Rumba Application Development Kit is a complete development environment that empowers Visual Basic developers to change how users interact with PC and host applications, combining advanced tools for creating connectivity links.

Rumba Tools for DDE and Rumba Tools for EHLLAPI

Rumba Tools for DDE and Rumba Tools for EHLLAPI enable advanced users to create simplified and transparent connectivity links between PCs and host computers. Rumba Tools for DDE allows Visual Basic applications to exchange data continuously with Rumba using DDE. Rumba Tools for EHLLAPI allows Visual Basic applications to exchange data with Rumba using EHLLAPI.

Libraries and Programming Utilities

Artisoft
6920 Koll Center Parkway
Suite 209
Pleasanton, California 94566
Contact: (415) 426-5355

Wired for Sound
DLL that can add sound capabilities to any Visual Basic form. Plays sound through PC speaker or sound boards. Includes API_SPEC.TXT file with code examples for Visual Basic programmers.

Black Ice Software, Inc.
Crane Road
Somers, NY 10589
Contact: (914) 277-7006
Fax: (914) 276-8418

TIFF SDK for Windows
A DLL that allows you to add TIFF 5.0 support to your Visual Basic applications, without learning the complexity of the Tagged Image File Format.

Crescent Software, Inc.
32 Seventy Acres
West Redding, CT 06890
Contact: Ethan Winer (203) 438-5300

PDQComm for Windows
PDQComm for Windows is a complete collection of routines that make it simple to add communications capabilities to programs written in Visual Basic.

QuickPak Professional
QuickPak Professional for Windows contains custom controls and a general purpose set of utilities for use with Visual Basic system. QuickPak Professional for Windows provides routines for quickly sorting and searching data, performing fast file operations, expression evaluation, and other useful tasks.

Data Techniques
1000 Business Center Drive Suite 120
Savannah, GA 31405
Contact: (912) 651-8003

Image Man
An object oriented Windows dynamic link library (DLL) that adds image display and print capabilities to applications. Supports TIFF, PCX, EPS, WMF, and BMP formats.

DemoSource
8646 Corbin Avenue
Northridge CA, 91324-4130
Contact: Brian L. Berman (800) 888-8063 Fax (818) 772-2877

DemoSource
A QuickLine voice library and VFEdit professional sound editor compatible with Visual Basic. This enables PCs to dispense prerecorded voice messages through standard touch-tone telephones for interactive mail order catalogs and automated outbound dialing systems for sales and telemarketing.

Eikon Systems Inc.
989 East Hillside Blvd, Suite 260
Foster City, CA 94404
Sales: (800) 727-2793
Contact: Jeff Galvin (415) 349-4664

Scrapbook+

Scrapbook+ is a Windows utility for managing Clipboard images, bitmaps, clip art, and other graphics. "Camera" tool allows you to create bitmap images of any portion of a screen. Can convert graphics between TIF, PCX, BMP, and MSP formats.

EMS Professional Shareware
4505 Buckhurst Ct.
Olney, MD 20832
Contact: (301) 924-3594 Fax (301) 963-2708

Public Domain Files

File collection of public domain and shareware file collections for Visual Basic programmers. Over 300 applications written in Visual Basic and utilities.

First Byte
19840 Pioneer Avenue
Torrance, CA 90503
Contact: Michael Belanger (310) 793-0600 x 212

Monologue for Windows

Make your Visual Basic applications talk! A text-to-speech utility implemented as a Windows DLL. Converts text into speech, to PC speaker or sound board.

Kofax Image Products
3 Jenner Street
Irvine, CA 92718
Fax: (714) 727-3144
Contact: Emily Backus (714) 727-1733

Kofax Image Processing Platform (KIPP)

KIPP comprises application-development software and controller boards, compatible with the Visual Basic system, that serve as the foundation for creating PC-based document image processing applications and systems.

MicroHelp, Inc.
4636 Huntridge Drive
Roswell, GA 30075-2012
Contact: Mark Novisoff (404) 594-1185

VB Tools

VB Tools is designed to add "pizazz" to Visual Basic programs. VB Tools includes numerous custom controls, graphics special effects, how to use Windows API services, utility modules, a program providing $INCLUDE capabilities and more.

MicroHelp Communications Library

MicroHelp Communications Library offers easy to use communications routines for Visual Basic invoked exactly like SubPrograms and Functions, including automatic file transfer routines using XModem, XModem CRC, YModem, YModem-Batch, ZModem, CompuServe B and ASCII transfers.

MicroHelp Muscle

MicroHelp Muscle is a library for the professional programmer that includes hundreds of assembly language routines and several high-level Visual Basic routines.

National Instruments
6504 Bridge Point Parkway
Austin, TX 78730-5039
Contact: Tim Dehne or Holly Matheny (512) 794-0100

NI-488.2 Windows Interface for Visual Basic

NI-488.2 Windows Interface for Visual Basic links a Visual Basic application to the NI-488.2 Windows GPIB driver software. System boards for the IEEE 488 interface available as well. Products connect Visual Basic with thousands of industry-standard programmable instruments.

NI-DAQ for Windows

NI-DAQ Windows Interface for Visual Basic applications using National Instruments' plug-in data acquisition boards. DLL with high-level data acquisition functions for developing data acquisition applications in Visual Basic.

Pinnacle Publishing
P.O. Box 8099
Federal Way, WA 98003
Contact: David Johnson (800) 231-1293 or (206) 941-2300

Graphics Server for Visual Basic

Graphics Server for Visual Basic is a custom control for integrating graphing and charting capabilities into Visual Basic applications. Include pie charts, bar charts and a variety of other graphs in 2D or 3D.

RealSound Inc.
4910 Amelia Earhart Drive
Salt Lake City, UT 84116
Fax (801) 359-2968
Contact: Janson Tanner (801) 359-2900

RealSound for Windows

A DLL for Windows providing an exciting enhancement to Visual Basic in hardware-quality digitized sound.

Scientific Software Tools, Inc.
30 East Swedesford Road
Malvern, PA 19355
Contact: Elise Furman (215) 889-1454, Fax (215) 889-1630

DriverLINX\VB

DriverLINX\VB is a high-performance data-acquisition engine for developing custom applications using Microsoft Visual Basic. Quickly create sophisticated virtual instruments that you could only dream of under DOS, in just days, using DriverLINX\VB. DriverLINX takes the form of a custom control that is added to the Toolbox of built-in Visual Basic controls.

The Stirling Group
172 Old Mill Road
Schaumberg, IL 60193
Contact: Viresh Bhatia, Managing Partner (708) 307-9197, (800) 3-SHIELD (800-374-4353)
Fax: (708) 307-9340

TbxSHIELD

A dynamic link library that allows you to create toolbox controls to include in you applications. Controls can be of any size, shape, or style, and can be created quickly and easily. Includes Visual Basic declarations and sample application.

Silicon Valley Products, Corp.
8 Paquatuck Avenue
East Moriches, NY 11940-0564
Contact: Paul Norris (516) 878-6438

QuickLine

A dynamic link library for use with the Visual Basic system to control TTI's telephone interface board for recording or playing messages, decoding telephone touch tones, and placement of calls.

VideoLogic
245 First Street
Cambridge, MA 02142
Contact: Karyn Scott (617) 494-0530

DVA-4000/ISA

VideoLogic's DVA-4000/ISA digital video adapter allows Visual Basic users to seamlessly integrate full-motion video with standard graphics and text in the Windows environment.

Ward Systems Group, Inc.
245 W. Patrick Street
Frederick, MD 21701
Contact: Marge Sherald (301) 662-7950

NeuroWindows

NeuroWindows is a neural network programming tool, designed to work with Microsoft's Visual Basic system, that builds powerful neural network applications which perform a wide variety of pattern recognition and prediction tasks.

WexTech Systems, Inc.
60 E. 42nd Street, Suite 1733
New York, NY 10165
Contact: (212) 949-9595
Fax: (212) 949-4007

Doc-to-Help

A Word for Windows 2.0 utility that allows you to create professional- quality documentation and automatically convert that documentation into Windows context-sensitive online help for your Visual Basic application. Includes the Microsoft Windows Help Compiler.

Graphics Utilities and Clip-Art

Dynalink Technologies
P.O. Box 593
Beaconsfield, Quebec
Canada H9W 5V3
Contact: (800) 522-4624

Clip'nSave 2.0 for Windows

Screen capture and image conversion program. Can capture any part of a screen to include in a Visual Basic program or print. Reads and writes mono, gray and color BMP, DIB, TIF, PCX, GIF and EPS files.

TechSmith Corporation
1745 Hamilton Road, Suite 300
Okemos, MI 48864
Contact: (517) 347-0800

SnagIt

Screen capture utilitiy for Windows. DDE support allows you to add screen capture capability to Visual Basic applications.

DDE Watch

Monitoring and debugging tool for dynamic data exchange

MicroCal, Inc.
22 Industrial Dr. E.
Northampton, MA 01060
Contact: (800) 969-7720.

Origin

Powerful scientific and technical graphics software for Windows. Supports DDE for plotting data from Visual Basic applications.

Publications and Services

Addison-Wesley Publishing Company, Inc.
1 Jacob Way
Reading, MA 01867
Orders: (800) 447-2226 or (617) 944-3700 or fax (617) 942-1117

Using Visual Basic By William Murray and Chris Pappas

Using Visual Basic is a hands-on guide to learning, using and mastering Visual Basic. It teaches the fundamentals of using Visual Basic and its programming elements. The book emphasizes how to design screens and place controls within Visual Basic. The authors lead readers through a series of applications that will serve as templates for applications that can be developed independently, including graphics, database, charting, and financial applications. The accompanying disk includes all of the applications in the book. (Currently available.)

Bantam Computer Books
666 Fifth Avenue
New York, NY 10103
Contact: Jono Hardjowirogo (212) 492-9826

Visual Basic Programming with Windows Applications By Douglas Hergert

A book oriented toward programmers with Basic experience interested in developing business solutions. (Currently available.)

Brady
Simon and Shuster, Inc.
15 Columbus Circle
New York, NY 10023
Sales: (800) 223-2348
Contact: Gene Smith (503) 639-9822

Visual Basic Written by Steven Holzner and The Peter Norton Computing Group--a complete introduction to Visual Basic.

(Currently available.)

The Cobb Group
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220
Sales: (800) 223-8720
Contact: Melissa Haeberlin (502) 491-1900

Inside Visual Basic

Inside Visual Basic is a 16-page monthly journal providing tips and techniques for using the Visual Basic programming system.

Cooper Software Inc.
3523 A Haven Avenue
Menlow Park, CA 94025-9986
Fax (415) 364-0593
Contact: Alan Cooper (415) 364-9150

QRC

Quick reference card for Microsoft Windows 3.0. Quick reference to all 597 Windows API calls.

ETN Corporation

RD4 Box 659
Montoursville, PA 17754-9433
Contact (Sales and Technical Information): (717) 435-2202

VB= mc^2: The Art of Visual Basic Programming
A book shipping in November, 1991 on programming in Visual Basic.

Fawcette Technical Publishing
299 California Ave, Suite 120
Palo Alto, CA 94306-1912
Contact: Jim Fawcette (415) 688-1808 Fax (415) 688-1812

Basic Pro
A bimonthly periodical for Basic professionals covering both text-mode and Windows Basic development issues. Provides advertising space for developers of Basic language products and add-on products, in addition to regular letters to the editor, guest columnist, product review, and upcoming industry event sections.

Microsoft Press
One Microsoft Way
Redmond, WA 98052-6399
Contact: Craig Johnson (206) 936-3895

The Microsoft Visual Basic Workshop
This book and software package is a one-stop source of imaginitive and useful Visual Basic system forms and subprograms to use in Microsoft Windows applications. (Currently available.)

Microsoft University
10700 Northrup Way
Bellevue, WA 98004-1447
(206) 828-1507

Microsoft University Visual Basic-- Advanced Topics course
A 3-day course covering concepts needed to write sophisticated event- driven, graphical programs and design applications that integrate with DDE and Windows DLLs.

National Instruments
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 338-9119, (800) 433-3488
Fax (512) 794-5569

Data Acquisition and Instrumentation Programming: A Look Through Windows 3.0
A seminar introducing attendees to programming data acquisition and instrument controls applications under Windows 3.0. Includes hands-on examples using Microsoft Visual Basic with DLLs, plug-in data acquisition boards, and GPIB controllers and instuments. Seminars will be offered in 48 different U.S. and Canadian cities in January through May 1992.

Osborne/McGraw Hill
2600 10th Street
Berkley, CA 94710
Sales: (510) 549-6614
Contact: Jeff Pepper (415) 549-6638

Visual Basic Inside and Out By Gary Cornell
A complete review of the Visual Basic programming system for Windows. (Available January, 1992.)

Programmer's Warehouse
8283 N. Hayden Road, Suite 195
Scottsdale, Arizona 85258
Contact: (800) 323-1809 or (602) 443-0580
Fax: (602) 443-0659

A full-service mail-order reseller for Visual Basic and all related companion products.

Que (Macmillan Computer Publishing)
11711 North College Avenue
Carmel, IN 46032
Tel. (800) 428-5331

Using Visual Basic

A book for beginning and intermediate programmers who want to write Visual Basic applications. Includes advanced features such as DDE, as well as a complete keyword reference section.

Sams (Macmillan Computer Publishing)
11711 North College Avenue
Carmel, IN 46032
Tel. (800) 628-7360

First Book of Visual Basic

A structured tutorial for the novice computer user covering the Visual Basic language and modern programming practice.

Waite Group Press
100 Shoreline Highway, Suite A-285
Mill Valley, CA 94941
Contact: (415) 331-0575
Order both books in advance for savings for 20%.

Visual Basic How-To

A book and disk package that contains hundreds of Visual Basic solutions from how to make an interface to how to use the Windows API functions. Ships January, 1992.

Visual Basic Super Bible

Explains each command, keyword, property, object and procedure of Visual Basic. 900 pages. All examples on disk. Ships March, 1992.