Extend Message Box Functionality

by Chris Barlow

Add to the already-powerful built-in MsgBox function.

Click & Retrieve Source CODE!

When you begin to write programs with Visual Basic, you'll find that you use the MsgBox function often. The message box is the easiest way to provide feedback to the user and request a simple response. When you understand its arguments, you can use the MsgBox function to handle many situations. But the really neat thing is that you can use the power of Visual Basic to extend the functionality of message boxes.

You only need to write this one line of code to display a small form with an OK button:

```
MsgBox "Order Saved"
```

The form's caption will be your Visual Basic project name, and your message will be centered on the form. After the user acknowledges the message by clicking on the OK button, your program continues with the next line of code.

If you need a simple "Yes/No" response from the user, you can call the same MsgBox function with additional arguments and save the return value:

```
result = MsgBox("Save this order?", vbYesNo, "Saving orders")
```

This line of code displays a form with two buttons captioned "Yes" and "No." The form's caption is "Saving orders," and the prompt is centered above the buttons. If the user clicks on the "Yes" button or presses the Enter key, the code returns the value 6 (equal to the intrinsic constant vbYes). Clicking on the "No" button returns the value 7 or vbNo. The neat thing about VB5 is that these constants pop up as choices in the editor so you don't have to remember them.

If you want to display a question mark icon to the left of your prompt on the message box, add the constant vbQuestion (32) to the second argument. Similarly, if you want the "No" button to be the default button for the Enter key, add the constant vbDefaultButton2 (256) to the second argument (see Figure 1):

```
result = MsgBox("Save this order?", vbYesNo + vbQuestion + _
   vbDefaultButton2, "Saving orders")
```

I developed a simple test form so you can experiment with the MsgBox functionality. This test application lets you experiment with all the arguments of the MsgBox function and see the values returned from the function call (see Listing 1). Premier Level members of The Development Exchange (DevX) can download this app from the Web (for details, see the Code Online box at the end of this column).

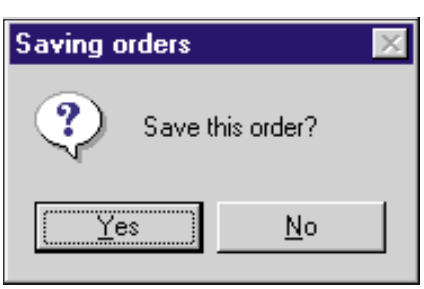
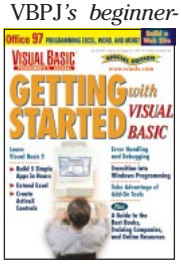


FIGURE 1 ***Simple MsgBox.*** *This standard message box built into Visual Basic is automatically modified by the button arguments that you pass in the function call.*

EXTEND THE MSGBOX

Although the built-in MsgBox function is powerful, you'll quickly find that you wish it could do more. For example, some of SunOpTech's applications that users execute on client workstations are also executed on servers as automatic daily processes. Normally, when the user runs these applications, we want all message boxes to display. However, when these applications are run on the server, it sets a public variable,

Chris Barlow is president and CEO of SunOpTech, a developer of manufacturing decision-support applications, including the ObjectBank, ObjectOrder, and ObjectJob Systems. Chris holds degrees from Harvard Business School and Dartmouth College, where he worked with Drs. Kemeny and Kurtz on the original Basic language. You can find a number of Chris' articles in VBPJ*'s beginner-oriented sister publication,* Getting Started with Visual Basic, *available on the newsstand. Reach Chris at ChrisB@SunOpTech.com or through SunOpTech's Web server at www.SunOpTech.com.*



gAutoMode, to True and doesn't display the message box. Instead, the server simply returns the default value for the function.

You could implement this functionality by finding every place that your application uses the MsgBox function and wrapping each line with an If statement to check the value of the gAutoMode variable, but this process would be extremely time consuming. You could make a global search-and-replace to change the MsgBox function to another function name such as MyMsgBox, but then you would have to remember to make this change with any new code added to the program.

A better method is to add a function to your application with the same name and arguments as VBA's MsgBox function. When Visual Basic compiles your application, it searches for references to function calls beginning with the functions in your application before it looks at the functions in the VBA engine. You can prove this by adding this function to your application:

```
Public Function MsgBox(prompt$, _
   Optional ButtonArg&, Optional title$)
VBA.MsgBox "You got into your routine"
End Function
```

Notice that you need to explicitly call VBA's MsgBox routine inside this procedure so that you don't keep calling your own MsgBox function over and over until you run out of stack space.

This ability to substitute your own function for one of VBA's functions can be powerful, but be careful. These substitutions can make it hard for another programmer to debug the application. For example, the VBA Trim function removes leading and trailing spaces from a string. If the last character is a Null—common in strings returned from a C++ DLL—then the spaces before the Null won't be removed. You can write your own Trim function that removes the Null characters and trailing spaces:

```
Function Trim(s$)
' trim string at first null
Dim P As Long
Trim = VBA.Trim(s$)
If Len(s$) Then
   P = InStr(s$, vbNullChar)
   If P Then
      Trim = VBA.Trim(Left$(s$, P - 1))
   End If
End If
End Function
```

However, if another programmer needs to debug your code (or you pick it up a few months later) and expects that the Trim function will leave Null characters at the end of the string, your application might fail if you forgot about your substituted Trim function.

VB5

```
Option Explicit
Private buttons&, icons&, DEFAULT&

Private Sub butDisplay_Click()
Dim prompt$, buttonarg&, title$, _
  Result%
buttonarg = buttons + icons + DEFAULT
prompt = Text1
title = Text2
If Len(title) Then
  Result = MsgBox(prompt, _
     buttonarg, title)
Else
  Result = MsgBox(prompt, buttonarg)
End If
VBA.MsgBox "Result was " & Result
End Sub

Private Sub Check1_Click()
gAutoMode = Check1.value * -1
End Sub

Private Sub Option1_Click(Index As _
  Integer)
buttons = Index
End Sub

Private Sub Option2_Click(Index As _
  Integer)
icons = Index * 16
End Sub

Private Sub Option3_Click(Index As _
  Integer)
DEFAULT = Index * 256
End Sub
```

LISTING 1 ***Test Form.*** *Notice how the Click events with the control arrays are used to set the arguments that will be passed to the MsgBox function. This form creates the call to the MsgBox function based on how you set the radio buttons on the form.*

Now that you understand the concept, you can expand your own message box function. First, add some code to your routine to properly call VBA's MsgBox function. Because the default for the buttons argument is zero, you can use Visual Basic 5's new feature to set a default value for an optional argument. Because you can only use a constant for a default argument, you need to use the IsMissing function to decide how to call VBA's MsgBox function:

```
Public Function MsgBox(prompt$, _
   Optional ButtonArg& = 0, _
   Optional Title$)
If IsMissing(Title) Or _
   Len(Title) = 0 Then
   MsgBox = VBA.MsgBox(prompt, ButtonArg)
Else
   MsgBox = VBA.MsgBox(prompt, _
      ButtonArg, Title)
End If
End Function
```

Your function will now pass along all calls to the intrinsic function. If you want to implement an automatic mode, you need to insert some code at the top of your function. First, check the gAutoMode variable to see if your application is running in automatic mode. If so, you don't need to display a message box; simply return the proper default value:

Buttons	DefaultButton1	DefaultButton2	DefaultButton3
OK	1	1	1
OK Cancel	1	2	—
Abort Retry Ignore	3	4	5
Yes No Cancel	6	7	2
Yes No	6	7	—
Retry Cancel	4	2	—

TABLE 1 ***By Default.*** *Determine the default value from the buttons argument by a GetDefaultReturn function that uses data to determine the return value.*

```
Public Function MsgBox(prompt$, _
   Optional ButtonArg& = 0, _
   Optional Title$)
If gAutoMode Then
   MsgBox = GetDefaultReturn(ButtonArg)
ElseIf IsMissing(Title) Or _
   Len(Title) = 0 Then
   MsgBox = VBA.MsgBox(prompt, ButtonArg)
Else
   MsgBox = VBA.MsgBox(prompt, _
      ButtonArg, Title)
End If
End Function
```

You can determine the default value from the buttons argument by a GetDefaultReturn function that uses data to determine the return value (see Table 1). This procedure uses Visual Basic's logical And statement to mask out the icon and default button portions of the buttons argument within a Select statement (see Listing 2).

Sometimes you need to add more functionality to the standard message box. The feature I need most is a fourth button. For example, I want to provide the user with choices of "Yes," "No," "Cancel," or "Yes for All." There's no easy way to accomplish

this with the built-in message box. You need to design your own form to replace the message box, which isn't a trivial task. The standard message box has a number of automatic features, such as automatic sizing of the form and the text, and centering of buttons, that take some work to duplicate in Visual Basic. Let's get started with a simple form that displays that desired fourth button.

Right-click on the Project window and add a form to your project. Add a command button at the bottom left of the form, set its Visible property to False, and size it to match the normal message box button sizes. Then copy it to the clipboard and paste it back on the form three times in a control array. Add a label control to contain the prompt text, and define public properties to hold the arguments and return value:

```
Public prompt$
Public title$
Public buttonarg&
Public button4cap$
Public ret%
```

In the Form Load event, add this code to set the form's caption and label:

```
Private Sub Form_Load()
If Len(title) = 0 Then
   Me.Caption = App.title
Else
   Me.Caption = title
End If
Label1.Caption = prompt
```

Set the caption for the fourth command button using the button4cap property:

```
If Len(button4cap) Then
   Command1(3).Caption = button4cap
   Command1(3).Visible = True
End If
```

Then set the captions on the command buttons and their Visible property based on the buttonarg property:

```
Select Case buttonarg And 7
Case 0  'OK
   Command1(0).Visible = True
   Command1(0).Caption = "OK"
Case 1  'OKCancel
   Command1(0).Visible = True
   Command1(0).Caption = "OK"
   Command1(1).Visible = True
   Command1(1).Caption = "Cancel"
Case 2  'AbortRetryIgnore
   Command1(0).Visible = True
   Command1(0).Caption = "Abort"
   Command1(1).Visible = True
   Command1(1).Caption = "Retry"
   Command1(2).Visible = True
   Command1(2).Caption = "Ignore"
```

(For the complete code, download Listing 3 from DevX.) Add code to the Click event of the command button control array to set the ret property with the proper return code:

```
Private Sub Command1_Click(Index As _
   Integer)
If Index = 3 Then
   ret = 8
Else
   Select Case Command1(Index).Caption
   Case "OK": ret = 1
   Case "Cancel": ret = 2
   Case "Abort": ret = 3
   Case "Retry": ret = 4
   Case "Ignore": ret = 5
   Case "Yes": ret = 6
   Case "No": ret = 7
   End Select
End If
Me.Hide
End Sub
```

Finally, add code to your MsgBox function to set the properties on your MyMsgBox form if the new Button 4 caption argument is filled in (download Listing 4 from DevX). Show this form with the vbModal argument so your program waits until the MyMsgBox form is hidden. Then get the return value and unload the form:

```
   MyMsgBox.buttonarg = buttonarg
   MyMsgBox.button4cap = But4
   MyMsgBox.prompt = prompt
   MyMsgBox.title = title
   MyMsgBox.Show vbModal
   MsgBox = MyMsgBox.ret
   Unload MyMsgBox
```

This new form is still pretty rough, but the basic functionality is in place. I'm sure you will think of additions, such as a Timer control so the message box form displays for 10 seconds and then selects the default button if the user doesn't click on another button. For more information on extending message box functionality, check out Francesco Balena's article, "Use the API to Find Hidden MsgBox Features" [*VBPJ* March 1997]. ×

VB5

```
Function GetDefaultReturn_
  (ButtonArg&) As Integer
Select Case ButtonArg And 7
Case 0  'OK
  GetDefaultReturn = 1
Case 1  'OKCancel
  If (ButtonArg And 256) = 256 Then
     GetDefaultReturn = 2
  Else
     GetDefaultReturn = 1
  End If
Case 2  'AbortRetryIgnore
  If (ButtonArg And 512) = 512 Then
     GetDefaultReturn = 5
  ElseIf (ButtonArg And 256) = _
     256 Then
     GetDefaultReturn = 4
  Else
     GetDefaultReturn = 3
  End If
Case 3  'YesNoCancel
  If (ButtonArg And 512) = 512 Then
     GetDefaultReturn = 2
  ElseIf (ButtonArg And 256) = _
     256 Then
     GetDefaultReturn = 7
  Else
     GetDefaultReturn = 6
  End If
Case 4  'YesNo
  If (ButtonArg And 256) = 256 Then
     GetDefaultReturn = 7
  Else
     GetDefaultReturn = 6
  End If
Case 5  'RetryCancel
  If (ButtonArg And 256) = 256 Then
     GetDefaultReturn = 2
  Else
     GetDefaultReturn = 3
  End If
End Select
End Function
```

LISTING 2 ***Determine the Default Value.*** *The GetDefaultReturn function uses Visual Basic's logical And statement to mask out the icon and default button portions of the buttons argument within a Select statement.*

Code Online

You can find all the code published in this issue of* VBPJ *on The Development Exchange (DevX) at http://www.windx.com. All the listings and associated files essential to the articles are available for free to Registered members of DevX, in one ZIP file. This ZIP file is also posted in the Magazine Library of the* VBPJ *Forum on CompuServe. DevX Premier Club members ($20 for six months) can get each article's listings in a separate file, as well as additional code and utilities for selected articles, plus archives of all code ever published in* VBPJ *and* Microsoft Interactive Developer *magazines.

Extend Message Box Functionality

Locator+ Codes

Listings ZIP file, including Listings 3 and 4 that don't appear due to space limitations (free Registered Level): VBPJ0797

✪ ***Listings for this article plus the complete test application (subscriber Premier Level): GS0797P***