**Implementing an Edit button in a**

# Database Front-End Applications using Visual Basic 3.0

**by Barth Riley, # 72677,3172**

I am currently writing a client and volunteer database application in Visual Basic 3.0. Recently, I wanted to implement an Edit button on my data-entry forms. Pressing the Edit button would allow the user to modify the data fields; prior to that point (and after the user saves the current record) the information on the screen would become read-only. This presented a problems, since Visual Basic offered no direct support for a read-only facility (e.g., viaa a ReadONly property). Setting the Enabled property to False was generally not satisfactory, since I wanted the user to be able to clearly see the contents of each field. Though setting the TabStop property to False prevents user access to a control via the Tab key, it does not prevent access through clicking on the control with the mouse, or pressing an access key. And lastly, despite indications to the contrary, setting the MultiLine property of a TextBox to True and the Enabled property to False does not achieve a read-only display of a TextBox's contents; it only made the contents of the text box invisible.

Implementing read-only controls required some doing, including a perusel of the Windows API. I also discovered that different types of controls require different methods to prevent the user from editing the contents of a control. The following details the methods I use to create read-only controls. Some of these techniques are fairly straightforward; others require brute force. I have also included a small demo project written in Visual Basic 3.0 to demonstrate these ideas. I hope you will find this helpful!

## Text Boexes:

Preventing user input in a text box is straightforward. Using a call to the Windows API function **SendMessage**, the following code fragment demonstrates the use of this function:

```
Declare Function SendMessage Lib "User" (Byval hWnd As integer,--
   Byval Msg As Integer, wParam as Integer, lParam As Any) --
   As Integer

Sub SetTBReadOnly(tbCtl  As TextBox, ByVal fReadOnly As Integer)
 Const WM_USER = &H400
 Const EM_SETREADONLY = WM_USER + 31
 Dim intRet As Integer

 intRet = SendMessage(tbCtl.hWnd, EM_SETREADONLY, fReadOnly,  0&)

End Sub
```

Thus, for example:

```
SetTBReadOnly Text1, True
```

would set the textbox Text1 to read-only mode.

Once a text box has been set to read-only mode, the control can still accept the focus (i.e., text will be highlighted when double-clicked) and respond to clicks and key presses, even though the contents of the text box remain unchanged. This can be advantageous if you wish to notify the user that s/he cannot edit the contents of the text box until the Edit button is pressed. Another advantage of this approach is that a text box can be activated/deactivated with the same subroutine. Unfortunately, this method can only be used with text boxes; there are no corollaries to the EM_READONLY message for other controls, including combo boxes, masked edit boxes, option buttons, and check boxes.

**Masked Edit Boxes:**

In my opinion, the easiest way to disable user input with a masked edit control (short of setting the Enabled property to False) is to set the Mask property to the contents of that field. For example, the following code fragment shows how a bound masked edit control could be made read-only:

```
Sub Data1_Validate(Action As Integer, Save As Integer)
    ' Clear the Mask to allow the value of the next
    ' record to be visible
    MaskEdit1.Mask=""
End Sub

Sub Data1_Reposition()
  If Not fEdit Then ' Disable user-input
    MaskEdit1.PromptInclude = True
    MaskEdit1.Mask = MaskEdit1.Text
  End If
End Sub
```

The Validate event handler for the data control clears the Mask property so that the contents of the control can be updated and visible. Once the new value for MaskEdit1 has been set, the Data1_Reposition subroutine will be triggered. The PromptInclude property is set to True so that the contents of the field will be visible. The Mask property is then set to the Text property. The Mask property is completely filled with literal characters, which prohibits the user from entering additional characters. Note that the fEdit flag indicates if editing operations are allowed. This approach works particularly well when the length of the mask is constant, such as with a social security number (e.g., "###-##-####")

or telephone number (e.g., "(###) ###-####"). With fields that contain variable-length data, such as numbers, you may need to experiment with different Mask values and set the PromptInclude property to False in order to display the proper value during Edit mode. To set a masked edit control back to edit mode, do the following:

```
Sub cmdEditBtn_Click()
   Dim szMask As String

   szMask = MaskEdiit1.Mask
   MaskEdit1.Mask = ""
   MaskEdit1.Text = szMask
   MaskEdit1.Mask = "##/##/##" ' in the case of a date field
End Sub
```

This subroutine simply saves the edit mask (the contents of the control), clears the edit mask, sets the Text property to the previous Mask property (szMask) and sets the Mask property again, this time to one which will accept user input.

**Combo Boxes:**

Exactly how one prevents the user from changing the contents of a combo box may depend in part on the Style of the combo box control. If the style is ComboList (2) setting the TabStop and Enabled properties to False will disable user input without graying the text of the current item. Because the combo box is disabled, however, no user events will be triggered when the user clicks on the control or presses a key. This may be undesirable if you wish to remind the user to press the Edit button to enable editing.

Alternately, if the style of the combo box is DropDown Combo (0) or Simple (1), or if you wish to respond to user events, a "brute force" approach is necessary. This approach requires code to handle the DropDown, KeyDown, and KeyPress events of the combo box, as follows:

```
Sub Combo1_DropDown()
   If Not fEdit Then
     ' Temporarily disable the control
     Combo1.Enabled = False
     Combo1.Enabled = True
   End If
End Sub

Sub Combo1_KeyDown (KeyCode As Integer)
   If Not fEdit Then KeyCode = 0
End Sub
```

```
Sub Combo1_KeyPress(KeyAscii As Integer)
   If Not fEdit Then KeyAscii = 0
End Sub
```

The first procedure (Combo1_DropDown) applies only to combo boxes which have a drop down list. When not in editing mode, (fEdit is False) the routine temporarily disables the combo box which effectively squelches the drop down operation (Though this causes a slight flicker within the combo box, the text remains visible; clever, eh?). The KeyDown and KeyPress event handlers prevent the user from scrolling through the list via the arrow keys and from typing a new value in the editing portion of the combo box. The fEdit flag should be declared as a form-level variable.

**Check Boxes and Option Buttons:**

There is no good way to prevent a user from changing the value of a check box or option button group. The only way I have found is to set the value of a check box or option button to its original value in response to a Click event. If anyone has a better approach, please let me know!

The following code demonstrates my method:

```
Sub Check1_Click()
Static OldValue As Integer

   If fEdit or fLoading Then
      OldValue = Check1.Value
   Else
     Beep
     Check1.Value = OldValue
   End If
End Sub
```

Because you are resetting the value of the control after the fact, the value will momentarily change when the user clicks on the control. You could write code to temporarily disable the control during a MouseDown event (similar to the code I wrote to prevent a combo box from dropping down). Unfortunately, this strategy is not reliable and does not prevent the user from changing the value of check box or option button by double clicking the mouse. Finally, you could disable the control or control group altogether. Alternatives anyone?

**Sample Code:**

I have included in the ZIP file a sample project demonstrating the use of an Edit button and read-only controls. The project (EDITDATA.MAK) includes one form (EMPLOYEE.FRM) and one module (EDITDATA.BAS). Since the

project includes masked edit control, it requires the Professional edition of Visual Basic. If you do not have the Professional edition, simply delete all references to the masked edit controls. The program references a database file called EMPLOYEE.MDB which is located in the C:\VB directory. Therefore the database files (EMPLOYEE.MDB and EMPLOYEE.LDB) should reside in this directory.

The program code is copyrighted. You are free to distribute the code in its original form only. If any of the code is incorporated in other programs, references to its author (Barth Riley) should be included with the ported code.

I hope you have found these notes and sample code to be helpful. If you have questions or comments, please drop me a line. My CompuServe number is 72677,3172. Happy coding!