Access Answers: What a drag it is (getting old?), Part 1

Doug Steele

Doug tries to address commonly asked questions from Access developers. This month, he looks at how to add drag-and-drop to an Access application.

Can I add drag-and-drop capabilities to an Access application?

The ability to drag-and-drop is so easy to implement in Visual Basic, but the Access form model is different, and it's not nearly as easy to implement drag-and-drop in Access. However, it is possible to do, although you need to control it all manually.

Just to be perfectly clear, I'm talking about dragging and dropping data, not the controls themselves. That means that some controls are not conducive to drag and drop. For example, you can't drag a Command button or a Toggle button. As well, some controls are mutually incompatible for dragging and dropping. While you might be able to drag a check box, what would you expect to happen if you dropped it on a list box? On the other hand, if you dragged a check box to a text box, you might want the text box to display True or False, depending on the state of the check box when you dragged it. Rich-text boxes already support drag-and-drop, so I'm going to ignore them.

Let's consider what makes up a drag-and-drop event. First, you need to detect that the drag has started. Once you've got a drag operation underway, you need to be able to detect when (and where) the drag has stopped. If the drag stopped somewhere that can accept a drop, you need to detect that fact. Finally, if you've detected a drop, you need to handle the drop event. Microsoft has KB articles that demonstrate one way to implement these events: http://support.microsoft.com/?id=137650 for Access 95/97, http://support.microsoft.com/?id=233274 for Access 2000 and http://support.microsoft.com/?id=287642 for Access 2002 (although the code is identical in each article). In this column, I'm going to extend how the implementation of that functionality.

Dragging something requires that the mouse be depressed while dragging. This means that to be able to detect when a drag has started, you can utilize the **`MouseDown`** event for each control from which you want to be able to drag. Even if you're actually not going to drag from the control when you activate the **`MouseDown`** event, there's no problem with initializing whatever's required, just in case.

To be able to detect when a drag has stopped, use the **`MouseUp`** event for each control from which you want to be able to drag. (If a mouse button is pressed while the pointer is over a control, that control receives all mouse events up to and including the last **`MouseUp`** event, regardless of where the mouse pointer actually is when the mouse button is released.).

The actual code you need to add to the **`MouseDown`** and **`MouseUp`** events of each control you want to be capable of being dragged is pretty simple. When the **`MouseDown`** event occurs, you want to set global references to the control itself, and to the form on which the control exists, as well as set a flag to indicate that a Drag has started. I use 3 module-level variables:

- **`mfrmDragForm`** (the form from which the value is being dragged)
- **`mctlDragCtrl`** (the control on mfrmDragForm from which the value is being dragged)
- **`mbytCurrentMode`** (a flag indicating whether the current action is Dragging, Dropping or nothing)

The easiest way I know of doing this is to have a **`DragStart`** routine, which can be called from each **`MouseDown`** event. In the VBA code associated with your form, you'll need something like:

```
Private Sub Text1_MouseDown (Button As Integer, _
  Shift As Integer, X As Single, Y As Single)

    Call StartDrag(Me)

End Sub
```

And in a module, you'll have something like:

```
Sub StartDrag (SourceForm As Form)

  Set mfrmDragFrm = SourceForm
  Set mctlDragCtrl = Screen.ActiveControl
  mintCurrentMode = DRAG_MODE

End Sub
```

(NOTE: You shouldn't use `Screen.ActiveForm` in place of `SourceForm` because you may be dragging from a subform. It was a somewhat arbitrary decision on my part not to pass the active control as a parameter, based largely on the fact that it means less typing in each `MouseDown` event. If it makes you feel better, you can pass the control as well!)

I like to take this one step further, though, since I consider it a good idea to give the user visual feedback. I want to change the mouse cursor to indicate that a drag is occurring. Without going into the actual code used to change the cursor (I use code similar to that at http://www.mvps.org/access/api/api0044.htm, although I name my functions `SetMouseCursor` and `SetMouseCursorFromFile`, rather than `MouseCursor` and `PointM`), I use a different icon depending on whether I'm dragging a single value, or multiple ones. This means that I need to be able to detect which is the case. To do this, I add a 4th variable to what's set in `StartDrag`: `mbytDragQuantity` (a flag to indicate whether we're dragging a single value, or multiple values). I then have a function `SetDragCursor` that uses that variable to determine which icon to use for the mouse cursor. Remembering that out of the standard Access controls, only the list box supports multi-selection, that means that the code for `StartDrag` actually looks more like:

```
Sub StartDrag(SourceForm As Form)

  Set mfrmDragForm = SourceForm
  Set mctlDragCtrl = Screen.ActiveControl
  mbytCurrentMode = DRAG_MODE
  If TypeOf mctlDragCtrl Is ListBox Then
    If mctlDragCtrl.ItemsSelected.Count > 1 Then
      mbytDragQuantity = MULTI_VALUE
    Else
      mbytDragQuantity = SINGLE_VALUE
    End If
  Else
    mbytDragQuantity = SINGLE_VALUE
  End If
  SetDragCursor

End Sub
```

(If you're using other controls that support multi-select, you'll need to add additional cases in the `TypeOf` check)

To detect when the dragging stops, have a `StopDrag` event that you can call from the `MouseUp` event of every control from which you want to be able to drag:

```
Private Sub Text1_MouseUp (Button As Integer, _
  Shift As Integer, X As Single, Y As Single)

    Call StopDrag

End Sub
```

`StopDrag` looks something like:

```
Sub StopDrag()

  mbytCurrentMode = DROP_MODE
  mbytDragQuantity = NO_MODE
  msngDropTime = Timer()
  SetDragCursor

End Sub
```

In addition to resetting the mode from **DRAG_MODE** to **DROP_MODE**, the drag quantity from either **SINGLE_VALUE** or **MULTI_VALUE** to **NO_MODE** and resetting the mouse cursor, it also sets a variable **msngDropTime** to the value of the built-in **Timer** function. This is important, since it's used in the next procedure to be called, **DetectDrop**.

Once you know that the dragging has stopped, you need to determine whether the drag ended on a control capable of accepting a drop. As soon as the **MouseUp** for the previous control has been handled, the **MouseMove** event of the new control should fire. That means that if you want a control to be capable of accepting a drop, you should be able to use the **MouseMove** event of that control to invoke the **DetectDrop** procedure:

```
Private Sub Text2_MouseMove(Button As Integer, _
  Shift As Integer, X As Single, Y As Single)

    Call DetectDrop(Me, Me!Text2, Button, Shift, X, Y)

End Sub
```

The **DetectDrop** procedure itself is a little more complex than the two others I've shown you so far:

```
Sub DetectDrop(DropForm As Form, DropCtrl As Control, _
               Button As Integer, Shift As Integer, _
               X As Single, Y As Single)

' If a drop hasn't happened, then exit.
  If mbytCurrentMode <> DROP_MODE Then
    SetDragCursor
    Exit Sub
  Else
    mbytCurrentMode = NO_MODE

  If Timer - msngDropTime > MAX_DROP_TIME Then
      Exit Sub
    Else
' Did we drag/drop onto ourselves?
      If (mctlDragCtrl.Name <> DropCtrl.Name) Or _
         (mfrmDragForm.hWnd <> DropForm.hWnd) Then
' If not, then a successful drag/drop occurred.
          Call ProcessDrop(mfrmDragForm, mctlDragCtrl, _
                           DropForm, DropCtrl, _
                           Button, Shift, X, Y)
      End If
    End If
  End If

End Sub
```

The first thing done is to check whether a Drop has occurred. I do this by checking whether **mbytCurrentMode** has been set to **DROP_MODE** by **StopDrag**. If it has, I then check to make sure that this invocation of **DetectDrop** was called by the **MouseMove** event that immediately followed the **MouseUp** event that invoked **StopDrag**. While I'm sure there are other ways of doing this, I find that comparing the results of the **Timer** function to the value of **msngDropTime** set by **StopDrag** is effective. If this is the appropriate invocation, I check that we haven't just dropped onto ourselves (this is necessary for those controls that are set up for both dragging from and dropping to). Note the use of the **hWnd** properties when comparing the two saved Form references. This is to be able to handle those situations where there are multiple instances of the same form open. It's possible that you might be trying to drag from a certain control on instance one of the form to the same control on instance two, so you can't just rely on the name of the form.

Once it's known that a drag-and-drop sequence has occurred, the last remaining thing to do is handle it. As you've probably guessed, this can be the most complicated part, especially when you allow dragging from controls which support multiselected values. As I alluded to earlier, you may have to make decisions about what controls can drag to which other controls, as well as decisions about what to do if you drag multiselected values onto controls that are only capable of showing a single value.

The simplest form of the **ProcessDrop** routine is something like:

```
Sub ProcessDrop(DragForm As Form, _
                DragCtrl As Control, _
                DropForm As Form, _
                DropCtrl As Control, _
                Button As Integer, _
                Shift As Integer, _
                X As Single, _
                Y As Single)

  DropCtrl = DragCtrl

End Sub
```

In other words, copy the current value of the control referenced by **DragCtrl** to the control referenced by **DropCtrl**. (**Value** is the default property for most controls. I suppose that to be completely correct, I should have used **DropCtrl.Value = DragCtrl.Value**) In actual practise, though, it's seldom that simple. If **DragCtrl** is a multiselect list box, for example, then as a bare minimum, you need to process each selected entry in that list box. If you're dragging to a multiselect list box to a text box, you might want to concatenate each of the selected entries:

```
Sub ProcessDrop(DragForm As Form, _
                DragCtrl As Control, _
                DropForm As Form, _
                DropCtrl As Control, _
                Button As Integer, _
                Shift As Integer, _
                X As Single, _
                Y As Single)

Dim strSelectedItems As String
Dim varCurrItem As Variant

  If TypeOf DragCtrl Is ListBox Then
    If DragCtrl.ItemsSelected.Count > 0 Then
      For Each varCurrItem In DragCtrl.ItemsSelected
        strSelectedItems = strSelectedItems & _
                    DragCtrl.ItemData(varCurrItem) & ", "
      Next varCurrItem
      If Len(strSelectedItems) > 2 Then
        strSelectedItems = Left$(strSelectedItems, _
                            Len(strSelectedItems) - 2)
      End If
      DropCtrl = strSelectedItems
    Else
      DropCtrl = DragCtrl
    End If
  Else
    DropCtrl = DragCtrl
  End If

End Sub
```

Here, if **DragCtrl** is a listbox with more than one row selected, I loop through all of the items in the ItemsSelected collection of that list box, concatenating each value to a string, and then assign the value of that string to the **DropCtrl**. Note that if the bound column of the listbox isn't the value you want to display, you'll have change the line value **DragCtrl.ItemData(varCurrItem)** to something more appropriate, such as **DragCtrl.Column(2, varCurrItem)**. However, if **DropCtrl** is another listbox, maybe what you want to do is copy (or move) the selected items from the source listbox to the target listbox. How you do this, of course, depends on how you populated the listboxes. The accompanying download database has a sample where I demonstrate how to drag from one listbox to another. In this case, the two list boxes are based on a table which has a Selected field in it. One list box represents those records in the table for which the Selected field is **False**, while the other list box represents those records for which the Selected field is **True**. This means that **ListBoxExample** (which I call from **ProcessDrop**) must be able to update the table, and requery both listboxes:

```
Sub ListBoxExample(DragForm As Form, _
```

```
                DragCtrl As Control, _
                DropForm As Form, _
                DropCtrl As Control, _
                Button As Integer, _
                Shift As Integer, _
                X As Single, _
                Y As Single)

Dim dbCurr As DAO.Database
Dim strSQL As String
Dim strMessage As String
Dim strWhere As String
Dim varCurrItem As Variant

  Set dbCurr = CurrentDb()

  strSQL = "UPDATE Customers SET Selected=" & _
    IIf(DragCtrl.Name = "lstListBox1", "True", "False")

  If (Shift And acShiftMask) = 0 Then
    If DragCtrl.ItemsSelected.Count > 0 Then
      For Each varCurrItem In DragCtrl.ItemsSelected
        strWhere = strWhere & "'" & _
                   DragCtrl.ItemData(varCurrItem) & _
                   "', "
      Next varCurrItem
      If Len(strWhere) > 2 Then
        strWhere = " WHERE [CustomerID] IN (" & _
          Left$(strWhere, Len(strWhere) - 2) & ")"
      End If
    Else
        strWhere = " WHERE [CustomerID] = '" & _
          DragCtrl & "'"
    End If
  End If

  If Len(strWhere) > 0 Then
    strSQL = strSQL & strWhere
  End If

  dbCurr.Execute strSQL, dbFailOnError

  DragCtrl.Requery
  DropCtrl.Requery

End Sub
```

Here, I check from which of the two list boxes I'm dragging. If I'm dragging from lstListBox1 (to lstListBox2), I know that I need to change the dragged records from not selected to selected. If I'm dragging from lstListBox2, I know I need to change them to not selected.

You may have noticed that in this case I'm using one of the other values passed to the routine from the MouseMove event, specifically the Shift value. This allows me to add the feature that if you drag from one box to the other while holding down the Shift key, all of the records are dragged, not simply the one(s) you've actually selected. (It also allowed me to justify why I'm passing those values from the **`MouseMove`** event to the **`DetectDrop`** routine, and then to the **`ProcessDrop`** routine!) The code **`If (Shift And acShiftMask) = 0`** is how I check to determine whether or not the Shift key is depressed: that expression will be non-zero if the shift key is depressed when the mouse is dragged. If it was depressed, I don't bother with a **`WHERE`** clause in my SQL statement: I simply change all the Selected values to either **`True`** or **`False`**. If the shift key is not depressed, I loop through the list of all selected rows in the list box (using the list box's ItemsSelected collection) and add each one to the **`WHERE`** clause.

Once I've created my SQL string, I execute it. (Aside: I feel that using the **`Execute`** method of the DAO **`Database`** object to run an SQL statement is better than using the **`DoCmd.RunSQL`** approach because it doesn't issue the "You're about to update n records…" message box, plus it allows you to trap any errors that may occur running the SQL.)

Now that I've updated the table appropriately, I requery the two list box controls, so that their content reflects the updated table.

Other dragged controls will require still different handling, and how you handle each dragged control may depend on what the drop control is.

For example, if you want to be able to drag check boxes to text boxes, presumably what you'd want appearing in the text box is "True" or "False". On the other hand, if you drag a check box onto another check box, you probably would want the drop check box to take on the same value as the dragged one. You can accomplish that using code like:

```
  If TypeOf DragCtrl Is CheckBox Then
' Assume we only allow dropping check boxes onto text boxes or check boxes
    If TypeOf DropCtrl Is TextBox Then
      DropCtrl = IIf(DragCtrl, "True", "False")
    ElseIf TypeOf DropCtrl Is CheckBox Then
      DropCtrl = DragCtrl
    Else
    End If
  End If
```

I'll give one more example. If you have an OptionGroup on your form, it will have a numeric value associated with it. You could drag that numeric value to a textbox, or you could determine the text associated with the selected option, and drag that text. In your code in ProcessDrop, you would have to specifically determine which textbox is to get just the number, and which is to get the text, using code similar to:

```
  If TypeOf DragCtrl Is OptionGroup Then
' Assume we only allow dropping option groups onto text boxes
    If TypeOf DropCtrl Is TextBox Then
      Select Case DropCtrl.Name
        Case "txtTextBox1"
          DropCtrl = DragCtrl
        Case "txtTextBox2"
          DropCtrl = ReturnSelectedOption(DragCtrl)
        Case Else
      End Select
    Else
    End If
  End If
```

where `ReturnSelectedOption` is something like:

```
Function ReturnSelectedOption( _
  OptionGroup As OptionGroup) As String

Dim ctlCurr As Control
Dim booGetText As Boolean
Dim strSelected As String

  For Each ctlCurr In OptionGroup.Controls
    If TypeOf ctlCurr Is OptionButton Or _
       TypeOf ctlCurr Is CheckBox Then
' Option Buttons and Check Boxes have labels
' associated with them. We need to determine
' which one is selected, and then determine
' the label associated with it.
      If ctlCurr.OptionValue = OptionGroup.Value Then
        strSelected = ctlCurr.Name
        booGetText = True
        Exit For
      End If
    ElseIf TypeOf ctlCurr Is ToggleButton Then
' Toggle Buttons has captions on them. Once we
' determine which one is selected, we then just
' need to determine its caption.
      If ctlCurr.OptionValue = OptionGroup.Value Then
        ReturnSelectedOption = ctlCurr.Caption
```

```
        booGetText = False
        Exit For
      End If
    End If
  Next ctlCurr

  If booGetText Then
' For each label on the Option Group, determine its
' parent control's name and compare it to the name of
' the selected control.
    For Each ctlCurr In OptionGroup.Controls
      If TypeOf ctlCurr Is Label Then
        If ctlCurr.Parent.ControlName = strSelected Then
          ReturnSelectedOption = ctlCurr.Caption
          Exit For
        End If
      End If
    Next ctlCurr
  End If

End Function
```

Hopefully you'll be able to take these various building blocks and combine them into a module that will meet your specific needs.

Note that I've only addressed how to drag and drop from one control to another control in the same Access application. Next month, we'll take a look at what can be done to drag from non-Access applications to controls on Access applications.

Doug Steele has worked with databases, both mainframe and PC, for many years. Microsoft has recognized him as an Access MVP for his contributions to the Microsoft-sponsored newsgroups. Check http://I.Am/DougSteele for some Access-related links. You can reach him at AccessHelp@rogers.com, but please note that personal replies are not guaranteed. However, ***please*** *don't hesitate to send ideas for future columns or, even better, complete columns!*