Drag'n'Drop – die vergessene Fähigkeit in Access

Eigentlich könnte alles ganz einfach sein: Um einen Eintrag von einer Liste in eine andere ziehen zu können, schaltet man die beiden jeweiligen Listboxen in den Drag- bzw. Drop-Modus, ergänzt zwei oder drei Ereignisse und ist fertig. So ist es in den übrigen MS-Office-Programmen wie beispielsweise Word, welche gemeinsam die MS-Forms-2.0-Bibliothek einsetzen.

Nicht so in Access. Die Umsetzung einer der typischsten Benutzer-Aktionen seit Erfindung der Computer-Maus, nämlich das Ziehen eines Objekts auf ein anderes, wird hier zu einer Odyssee durch komplizierteste Programmier-Tricks. Dabei sind die Anforderungen eigentlich banal:

- Eine (oder mehrere) bereits markierte Elemente des Startobjekts werden mit der Maus bei gedrückter linker Taste gezogen …
- … und der Mauscursor zeigt während der Bewegung automatisch, ob das überfahrene Objekt dropfähig wäre, …
- … bis beim Loslassen der Maus über einem geeigneten Zielobjekt die gezogenen Elemente durch VBA-Code nach Belieben eingefügt werden.

Das kann eigentlich nicht so schwierig sein? Doch. Aber immerhin können wir uns auf einige wenige Elemente und Anlässe beschränken.

Listbox und Textbox auf Formularen

Als Controls werde ich nur Listbox- und Textbox-Controls auf Formularen betrachten. Es ist technisch möglich, eine Checkbox auf eine andere zu ziehen und dadurch deren Wert zu übernehmen. Aber was soll das? Viel schneller lässt sich deren Wert ohnehin durch einen direkten Klick umschalten, das lohnt also den Aufwand nicht. Ebenso mag ein Button auf einen anderen gezogen werden können, aber auch da lässt sich der Sinn einer solchen Benutzerführung nicht erklären.

Ohnehin kommen nur Formulare in Betracht, weil auf ausgedruckten Berichten ja keine Maus-Interaktion mehr stattfinden kann. Schließlich wird es dabei nur um die Verschiebung der darin enthaltenen *Daten*, nicht der *Objekte* gehen.

Daten innerhalb/Dateien von außen

Viele Daten werden sich also zwischen zwei Controls innerhalb von Access bewegen. Dazu kommt aber noch ein Sonderfall, bei dem per Drag'n'Drop Dateien aus dem Windows-Explorer in ein Access-Control gezogen werden, um dort Dateinamen anzugeben. Das geht zwar auch "aktiv" durch Aufruf eines DateiÖffnen-Dialogs, das Ziehen aus einem bereits geöffneten Explorer-Fenster ist aber intuitiver und schneller.

Umsetzung

Es gibt nicht eine einzige Lösung, die perfekt ist. Wir werden uns also mit den Vor- und Nachteilen der verschiedenen Umsetzungen beschäftigen müssen:

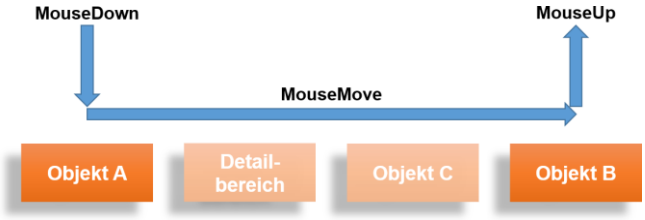
- ***MouseXXX*-Ereignisse**: Die verschiedenen `MouseDown`-, `MouseMove`- und `MouseUp`-Ereignisse sind die typischen Methoden, um das Ziehen und Verschieben mit der Maus abzufangen und via VBA darauf zu reagieren.
- **Funktionen**: Statt die Maus-Ereignisse als Prozeduren mit standardisierten Namen aufzurufen, können in der jeweiligen Ereignis-Eigenschaft eines Controls auch Funktionen mit Parametern aufgerufen werden.
- **SubClassing**: Was Access nicht direkt als Ereignis anbietet, lässt sich mit dieser Technik hinzufügen. Dabei werden fast beliebige zusätzliche Ereignisse abgefangen.
- **Klasse mit *event sink***: Die Ereignisse der einzelnen Controls können viel kompakter auch mit `WithEvent`-Befehlen in einer Klasse abfangen werden.
- **APIs**: Access lässt sich durch Aufruf der Windows-APIs (Application Programming Interface) um fehlende Funktionalitäten erweitern, mit denen die Maus gesteuert oder Informationen abgefragt werden können.

- **Kein Access**: Durch die Einbindung weiterer Libraries greift Access nicht nur auf ergänzende Fähigkeiten zu, sondern kann auch beispielsweise die Steuerelemente der übrigen Office-Programme nutzen, die sich technisch deutlich von ihren Access-Geschwistern unterscheiden.

Ich kann schon jetzt versprechen, dass es spannend bleibt. Ich kann hingegen nicht versprechen, dass es einfach wird. Aber der Reihe nach…

MouseXXX-Ereignisse

Beim Drag'n'Drop zwischen zwei Controls eines Formulars braucht es nur drei Schritte, die alle in `MouseXXX`-Ereignissen abgebildet werden können:



Schematischer Ablauf einer Drag'n'Drop-Aktion.

Im Start-Control *Objekt A* drückt der/die Benutzer/in die linke Maustaste (`MouseDown`), bewegt dann die Maus mit immer noch gedrückter Taste (`MouseMove`) zum Ziel-Control *Objekt B* und lässt die Taste dort los (`MouseUp`). Zwischendurch könnten im `MouseMove` noch andere Objekte oder Bereiche überfahren werden, was aber ignoriert werden soll.

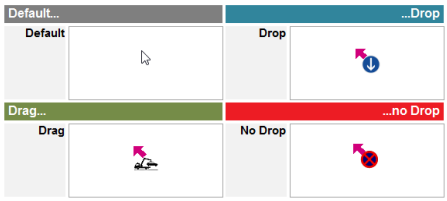
Dazu passen die vorhandenen VBA-Ereignisse, welche sogar noch zusätzliche Informationen wie die benutzten Tasten (`Button` für Maustaste und `Shift` für die Umschalttaste) oder die x-/y-Koordinaten zurückgeben:

```
Private Sub ObjektA_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
    'Drag startet mit diesem Objekt
End Sub

Private Sub ObjektXXX_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
    'Move läuft über beliebige andere Objekte hinweg
End Sub

Private Sub ObjektB_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
    'Drop findet hier statt
End Sub
```

Um das Geschehen optisch deutlich zu machen, soll der Cursor laufend anzeigen, in welchem Modus er sich gerade befindet, wenn ein Drag'n'Drop von *Objekt A* nach *Objekt B* stattfindet:

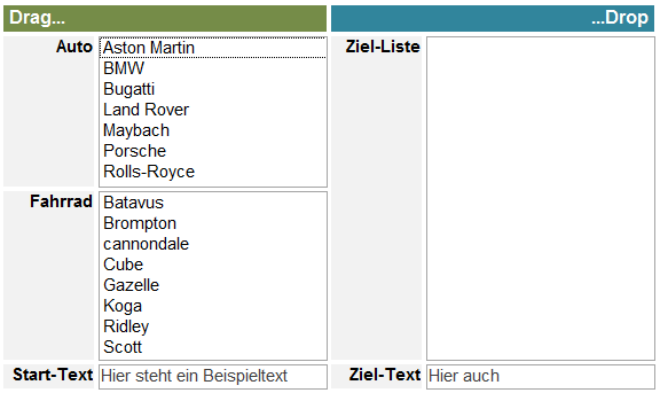


Standard-Cursor (oben links) und Cursor-Icons für Drag-Modus (unten links), für Drop-fähige Controls (oben rechts) und für Bereiche ohne Controls (unten rechts)

Der Einfachheit halber wird keine konkrete Drop-Aktion stattfinden, sondern lediglich eine einfache `MsgBox` erscheinen, von woher nach wohin Daten verschoben/kopiert würden. Wenn der/die Benutzer/in nun über einem Drop-fähigem Control wie *Objekt B* die Maustaste loslässt, könnte der in `MouseUp` definierte VBA-Code dann die gewünschten Drop-Aktionen durchführen. Soweit die Theorie.

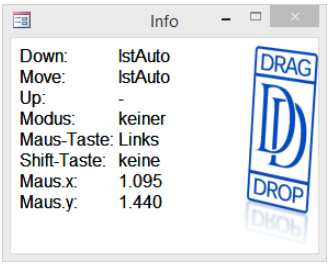
Umsetzung

Ein Access-Formular (*frmDnD_01_MouseXXX*) ohne Datenbindung enthält zum Testen und Vorführen mehrere Controls, bei denen von einer der linken Listboxen ein Wert auf die Ziel-Listbox rechts gezogen werden können soll:



Beispielformular mit zwei Start-Listboxen *lstAuto* und *lstFahrrad*, deren markierter Inhalt mittels Drag'n'Drop in die Ziel-Listbox *lstZiel* übernommen werden soll.

Außerdem existieren bereits diverse Prozeduren `LadeCursorDrag`, `LadeCursorDrop` etc. zur Anpassung des Cursor-Icons und um die Tasten- bzw. Mauskoordinaten (`ZeigeInfo`) oder einen Standardtext (`ZeigeStandardtext`) in einem PopUp-Formular anzuzeigen. Das hält den benötigten Code im Formularmodul möglichst kurz.



Info-PopUp-Formular mit den ermittelten Daten.

Trotzdem braucht das mit VBA-üblichen Ereignissen relativ viel Code, selbst wenn die gleichen Ereignisse jeweils in einer zentralen Prozedur zusammengefasst werden. Wesentliche Teile des Codes aus dem Formularmodul sind hier zu sehen, ähnliche Prozeduren habe ich hier weitestgehend weggelassen.

Damit auch beim `MouseUp` noch das Control bekannt ist, in welchem vorher das `MouseDown` stattgefunden hat, werden diese drei möglichen Objekte in modulöffentlichen Variablen verwaltet. Weil es sich tatsächlich nicht nur um Controls, sondern beispielsweise auch um einen Bereich (`Section`-Objekt) handeln kann, muss der Datentyp `Object` sein:

```
Dim m_objDown As Object
Dim m_objMove As Object
Dim m_objUp As Object

Private Sub Form_Close()
    ZeigeStandardtext
End Sub

'**********Liste Auto*******************
Private Sub lstAuto_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
    meinMouseDown Me.lstAuto, Button, Shift, X, Y
End Sub
Private Sub lstAuto_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
    meinMouseMove Me.lstAuto, Button, Shift, X, Y
End Sub
Private Sub lstAuto_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
    meinMouseUp Me.lstAuto, Button, Shift, X, Y
End Sub

'**********Liste Fahrrad, Zeile Starttext, Zeile Zieltext, Detailbereich, etc.*******************
'jeweils drei entsprechende Ereignisse wie bei Liste Auto
```

Dieser Block aus je drei `MouseXXX`-Prozeduren muss für jedes Objekt (hier also schon sechsfach) geschrieben werden. Das kostet nicht nur viel Zeit, sondern auch Platz. Im gleichen Formularmodul gibt es dann die allgemeinen Prozeduren, welche von den konkreten Ereignis-Prozeduren der einzelnen Objekte aufgerufen werden:

```
'**********Mouse-Ereignisse*******************
Private Sub meinMouseDown(ctlDieses As Object, Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    Set m_objDown = ctlDieses
    LadeCursorDrag
    ZeigeInfo m_objDown, m_objMove, m_objUp, m_intModus, Button, Shift, X, Y
End Sub
```


```
Private Sub meinMouseMove(ctlDieses As Object, Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    Set m_objMove = ctlDieses
    ZeigeInfo m_objDown, m_objMove, m_objUp, m_intModus, Button, Shift, X, Y
End Sub

Private Sub meinMouseUp(ctlDieses As Object, Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    Set m_objUp = ctlDieses
    LadeCursorDrop
    ZeigeInfo m_objDown, m_objMove, m_objUp, m_intModus, Button, Shift, X, Y
    DropMeldung m_objDown, m_objUp
    Set m_objUp = Nothing
    Set m_objDown = Nothing
    LadeCursorDefault
End Sub
```

So sollte grundsätzlich der Code aussehen. Leider hat dieser Code einen Haken: Er funktioniert nicht wie erwartet. Sowohl `MouseMove` als auch `MouseUp` liefern *nicht* das Control, über dem sich die Maus befindet, sobald die Maustaste gedrückt ist.

MouseMove/MouseUp-Ereignisse sind unbrauchbar

Das `MouseMove`-Ereignis liefert eigentlich die Daten des Objektes, über dem sich die Maus befindet. Das lässt sich leicht zeigen, weil die x-/y-Koordinaten hier im Info-Formular laufend angezeigt werden. Aber das stimmt nicht mehr, sobald in einem vorherigen `MouseDown` die Maustaste gedrückt wurde! Dann betrifft es immer noch das Start-Objekt, über welchem die Maus zu diesem Zeitpunkt ursprünglich stand.

Ebenso gilt das für `MouseUp`: Sobald die Maustaste gedrückt bleibt, was ja ein elementares Kennzeichen einer Drag'n'Drop-Aktion ist, liefert `MouseUp` keineswegs das Ziel-Objekt, sondern immer noch das Start-Objekt.

MouseXXX-Ereignisse fehlen bei manchen Controls

Ein vergleichsweise geringeres Problem besteht darin, dass Controls ohne optionalen Fokus (also vor allem Label-Controls) gar keine `MouseXXX`-Ereignisse besitzen. Um diese aber während der Mausbewegung im Cursor-Icon unterscheiden zu können, muss eine Reaktion auf `MouseMove` möglich sein.

Wirklich lästig ist aber, dass Labels, die Access-typisch mit einem Daten-Control wie Listbox oder Textbox verbunden sind, auch noch die Ereignisse dieses anderen Controls anzeigen. So täuscht ein verbundenes Label also sogar noch mit einem fremden Ereignis vor, es sei die zugehörige Text- oder Listbox.

Zusammenfassung

Die VBA-übliche Schreibweise der konkreten Ereignisprozeduren für jedes betroffene Control braucht nicht nur viel Platz und Zeit, sondern hat vor allem zwei K.O.-Kriterien: `MouseMove` und `MouseUp` liefern beim Ziehen mit gedrückter Maus nicht das Ziel-Control, sondern immer noch das Start-Control. Genau genommen ist das sogar überhaupt kein Fehler, sondern wird genau so in der VBA-Dokumentation angekündigt. Das hilft aber auch nicht weiter.

Es gibt Beispiel-Lösungen, bei denen statt des ungeeigneten `MouseUp` das anschließende nächste `MouseMove` genutzt wird. Weil dieses nicht mehr mit gedrückter Maustaste stattfindet, liefert es wieder das korrekte Ziel-Control. Das löst aber nicht das Problem mit `MouseMove`, so dass der Cursor beim Ziehen immer noch nicht anzeigen kann, ob er sich über einem Drop-fähigen Control befindet.

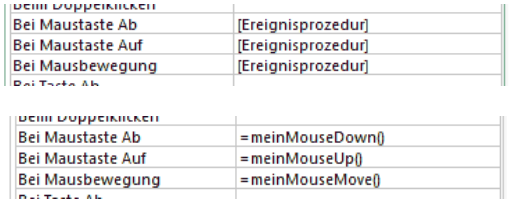
Vorteile	Nachteile
Einfache, VBA-übliche Schreibweise, automatische Rückgabe-Parameter	`MouseMove` und `MouseUp` liefern beim Ziehen falsche Controls, Controls ohne optionalen Fokus (Labels) können nie `MouseXXX`-Ereignisse senden, verbundene Labels senden Ereignisse des anderen Controls

Alternative Ereignis-Steuerung

Wenn also diese Art der Programmierung scheitert, liegt eine potentielle Lösung in einer unerwarteten Richtung. Es geht nicht darum, diese VBA-üblichen Schreibweise besser zu programmieren, sondern darauf zu verzichten. Es gibt nämlich noch andere Varianten, um ein Objekt mit einem Ereignis zu verbinden.

Funktionen statt Ereignis-Prozeduren

Die übliche Technik mit vordefinierten Ereignis-Prozeduren braucht für *jedes* betroffene Control drei `MouseXXX`-Prozeduren, selbst wenn die eigentlichen Aktionen sich zentral zusammenfassen lassen. Stattdessen lässt sich dort aber auch direkt eine Funktion aufrufen. Manuell entspräche das einem Wechsel der Control-Eigenschaften wie folgt:



Üblicher Aufruf der Ereignis-Prozeduren (oben) und alternativer Aufruf beliebiger Funktionen (unten)

Die `meinMouseXXX`-*Funktionen* ersetzen die im obigen Beispiel gezeigten gleichnamigen *Prozeduren*. Es müssen technisch bedingt Funktionen sein, auch wenn deren Rückgabewert schlicht ignoriert wird. Damit lassen sich diese Zuweisungen nun auch beim Laden des Formulars für beliebig viele Controls vornehmen:

```
Private Sub Form_Load()
    Dim ctlX As Control
    Dim intI As Integer

    On Error Resume Next    'wegen Labels o.ä.
    For Each ctlX In Me.Controls
        ctlX.OnMouseDown = "=meinMouseDown(""" & ctlX.Name & """," & typControl & ")"
        ctlX.OnMouseMove = "=meinMouseMove(""" & ctlX.Name & """," & typControl & ")"
        ctlX.OnMouseUp = "=meinMouseUp(""" & ctlX.Name & """," & typControl & ")"
    Next

    For intI = 0 To 4
        If Me.Section(intI).Visible Then
            Me.Section(intI).OnMouseDown = "=meinMouseDown(""" & Me.Section(intI).Name & """," & _
                typSection & ")"
            Me.Section(intI).OnMouseMove = "=meinMouseMove(""" & Me.Section(intI).Name & """," & _
                typSection & ")"
            Me.Section(intI).OnMouseUp = "=meinMouseUp(""" & Me.Section(intI).Name & """," & _
                typSection & ")"
        End If
    Next
    On Error GoTo 0
End Sub
```

Damit auch Bereiche (`Section`-Objekt) später einen passenden Maus-Cursor anzeigen und reagieren können, werden diesen ebenfalls die `MouseXXX`-Ereignisse zugewiesen (*frmDnD_02_Functions*). Die zentralen Programmierungen ändern sich nur minimal, weil in einer zusätzlichen Zeile das aufrufende Objekt umgewandelt werden muss:

```
'**********Mouse-Ereignisse*******************
Function meinMouseDown(strName As String, objTyp As enmTyp)
    Set m_objDown = ObjektMitTyp(Me, strName, objTyp)
    LadeCursorDrag
    ZeigeInfo m_objDown, m_objMove, m_objUp, unbekannt, 99, 99, 0, 0
End Function

Function meinMouseMove(strName As String, objTyp As enmTyp)
    Set m_objMove = ObjektMitTyp(Me, strName, objTyp)
    ZeigeInfo m_objDown, m_objMove, m_objUp, unbekannt, 99, 99, 0, 0
End Function

Function meinMouseUp(strName As String, objTyp As enmTyp)
    Set m_objUp = ObjektMitTyp(Me, strName, objTyp)
'wie bisher
End Function
```

Da in den Objekt-Eigenschaften der Aufruf solcher Ereignis-Funktionen nur als `String` gespeichert werden kann, sind auch für die Parameter nur `String`-Datentypen möglich. Daher enthält der zweite Parameter eine Zahl, anhand derer sich der Objekttyp rekonstruieren lässt:

```
Private Function ObjektMitTyp(frmMe As Form, strName As String, objTyp As enmTyp)
    Select Case objTyp
    Case typControl
        Set ObjektMitTyp = frmMe.Controls(strName)
    Case typSection
        Set ObjektMitTyp = frmMe.Section(strName)
    Case typForm
        Set ObjektMitTyp = frmMe
    End Select
End Function
```

Die Schreibweise mit zentralen Funktionen für die Ereignisse verkürzt den Standard-Code im Formularmodul erheblich. Wenn sinnvollerweise auch noch der Code in `Form_Load` und die Funktion `ObjektMitTyp` in einem allgemeinen Modul bereitgestellt wird, bleiben im Formularmodul nur noch die drei `meinMouseXXX`-Funktionen übrig.

Durch die Schleifen ist die Zuweisung flexibel und nicht auf eine bestimmte Anzahl von Controls begrenzt. Allerdings sind nur `String`-Parameter möglich und es gibt keine automatischen Rückgabe-Parameter über die Maustasten oder x-/y-Koordinaten.

Vorteile	Nachteile
Nur noch zentrale VBA-Funktionen notwendig, sehr viel Ereignis-Code entfällt, beliebig viele Controls automatisch nutzbar	Keine Rückgabe-Parameter mehr, Parameter können nur `String` sein

SubClassing

SubClassing ist die Technik, sich mit eigenen Meldungen zwischen Access und die Windows-Fenster-Verwaltung einzuklinken. Dabei werden die üblichen Befehle für Mausbewegungen, Fensterveränderungen etc. abgefangen oder manipuliert. Technisch benötigt das relativ wenig Aufwand, schon mit zwei API-Funktionen lässt sich der "Lauschangriff" starten:

```
Private Declare Function CallWindowProcA Lib "user32" _
    (ByVal lpPrevWndFunc As Long, ByVal HWnd As Long, ByVal Msg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long

Private Declare Function SetWindowLongA Lib "user32" _
    (ByVal HWnd As Long, ByVal nIndex As Long, ByVal wNewWord As Long) As Long

Const GWL_WNDPROC As Long = -4

Dim m_lngProc As Long
Dim m_lngHWnd As Long

Function BelauscheAccessFenster(ByVal HWnd As Long, ByVal Msg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
    BelauscheAccessFenster = CallWindowProcA(m_lngProc, HWnd, Msg, wParam, lParam)
End Function

Function StarteSubclassing() As Long
    m_lngHWnd = Application.hWndAccessApp
    m_lngProc = SetWindowLongA(m_lngHWnd, GWL_WNDPROC, AddressOf BelauscheAccessFenster)
End Function

Function BeendeSubclassing()
    SetWindowLongA m_lngHWnd, GWL_WNDPROC, m_lngProc
    m_lngProc = 0
End Function
```

Nachdem `StarteSubclassing` gestartet wurde, ermittelt `BelauscheAccessFenster` die Windows-Nachrichten des Access-Fensters und könnte darauf reagieren.

Allerdings gibt es ein wesentliches Problem dieser (ansonsten durchaus üblichen und verbreiteten) Technik: Access stürzt zuverlässig ab. Zwar nur, sobald der VBA-Editor geöffnet ist, aber das alleine reicht ja schon, weil sonst vor jedem Test das Programm komplett geschlossen werden müsste. Außerdem ist ohne VBA-Editor ein Debugging ziemlich unmöglich (siehe *SubClassing.accdb*).

Vorteile	Nachteile
Tiefer Eingriff in Windows-Nachrichten-System	Zuverlässiger Absturz von Access

DLL mit SubClassing

Die einzige Chance auf funktionierendes SubClassing bestünde darin, dieses in eine externe DLL zu verlagern. Dazu gibt es verschiedene Beispiele unter

http://www.donkarl.com/Downloads/AEK/AEK10_API.zip (AEK10, Jörg Ostendorp)

http://www.access-im-unternehmen.de/326 (André Minhorst)

http://www.cpearson.com/excel/SubclassingWithSSubTmr6.htm (vbaccelerator.com, Charles Pearson)

Da dies aber immer eine zusätzliche Programmiersprache benötigt und damit die reine Access-Lösung verlässt, soll es hier nicht weiter betrachtet werden. Zudem sind viele Kunden aus Sicherheitsbedenken nicht bereit, irgendeine DLL auf ihren Rechnern zu installieren.

Klasse mit *event sink*

Damit bleibt als weitere Variante eine Klasse mit Einsatz von `WithEvent`, bei der die Ereignisse von zugewiesenen Controls darin abgefangen und verarbeitet werden (eine so genannte *event sink*, engl. für "Ereignis-Senke"). Dadurch reduziert sich der Code im Formularmodul auf den Aufruf der Klasse `clsDragnDrop` plus eine Zeile je Control und sieht für das Beispielformular komplett so aus:

```
Private clsDD As clsDragnDrop

Private Sub Form_Load()
    Set clsDD = New clsDragnDrop
    With clsDD
        Set .SetSection01 = Me.Formularkopf
        Set .SetSection02 = Me.Detailbereich
        Set .SetSection03 = Me.Formularfuß

        Set .SetListCtl01 = Me.lstAuto
        Set .SetListCtl02 = Me.lstFahrrad
        Set .SetListCtl03 = Me.lstZiel

        Set .SetEditCtl01 = Me.edtStart
        Set .SetEditCtl02 = Me.edtZiel
        Set .SetEditCtl03 = Me.auto_ID

        .AlleEventsInitialisieren Me
    End With
End Sub

Private Sub Form_Close()
    Set clsDD = Nothing
End Sub
```

Um es übersichtlich zu halten, sind im weiter unten gezeigten Klassenmodul die Objekte *Section02*, *Section03*, *ctlEdit01*, *ctlEdit02*, *ctlEdit03*, *ctlList02*, *ctlList02* und *ctlList03* nicht oder nur teilweise dargestellt, da sie sich komplett identisch verhalten. Der durchaus noch erhebliche Schreibaufwand durch diese Wiederholungen ist nur einmalig in der Klasse zu leisten, deren wesentlicher Code vereinfacht so aussieht:

```
Private WithEvents Section01 As Access.Section    'Section02 bis ctlList03 hier nicht dargestellt
Private WithEvents ctlEdit01 As Access.TextBox
Private WithEvents ctlList01 As Access.ListBox

Dim m_frmDieses As Form
Dim m_objDown As Object
Dim m_objMove As Object
Dim m_objUp As Object
Dim m_intModus As enmModus

Private Sub Class_Initialize()
    m_intModus = enmModus.keinModus
End Sub

Private Sub Class_Terminate()
    ZeigeStandardtext    'stellt das Info-Formular auf neutralen Text zurück
End Sub

'Section02 bis ctlList03 hier nicht dargestellt:
Public Property Set SetSection01(secDiese As Access.Section): Set Section01 = secDiese: End Property
Public Property Set SetEditCtl01(ctlDiese As Access.TextBox): Set ctlEdit01 = ctlDiese: End Property
Public Property Set SetListCtl01(ctlDiese As Access.ListBox): Set ctlList01 = ctlDiese: End Property

Sub AlleEventsInitialisieren(frmMe As Form)
    Set m_frmDieses = frmMe

    MouseDownAktivieren Section01    'Section02 bis ctlList03 hier nicht dargestellt
    MouseMoveAktivieren Section01
    MouseUpAktivieren Section01
End Sub
```

```
Private Sub MouseDownAktivieren(objDieses As Object)
    If Not objDieses Is Nothing Then
        objDieses.OnMouseDown = "[Event Procedure]"
    End If
End Sub

Private Sub MouseMoveAktivieren(objDieses As Object)
    If Not objDieses Is Nothing Then
        objDieses.OnMouseMove = "[Event Procedure]"
    End If
End Sub

Private Sub MouseUpAktivieren(objDieses As Object)
    If Not objDieses Is Nothing Then
        objDieses.OnMouseUp = "[Event Procedure]"
    End If
End Sub

Private Sub Standard_MouseDown(objDieses As Object, Button As Integer, Shift As Integer, _
    x As Single, Y As Single)
    Set m_objDown = objDieses
    m_intModus = enmModus.DragModus
    ZeigeInfo m_objDown, m_objMove, m_objUp, 0, Button, Shift, x, Y, "von:", "clsDD"
End Sub

Private Sub Standard_MouseMove(objDieses As Object, Button As Integer, Shift As Integer, _
    x As Single, Y As Single)
    Set m_objMove = objDieses
    If m_intModus = enmModus.DropModus Then
        Set m_objUp = objDieses
    End If
    If TypeOf objDieses Is Section Or TypeOf objDieses Is Control Then
        MouseMoveErweitert
    End If
End Sub

Private Sub Standard_MouseUp(objDieses As Object, Button As Integer, Shift As Integer, _
    x As Single, Y As Single)
    m_intModus = enmModus.DropModus
    ZeigeInfo m_objDown, m_objMove, m_objUp, 0, Button, Shift, x, Y, "von:", "clsDD"
    MausPositionSchubsen
End Sub
```

Inhaltlich hat sich der Code gegenüber der Programmierung im Formular nur unwesentlich geändert, aber er muss jetzt nicht mehr für jedes Formular getrennt geschrieben werden. Da `WithEvent` nicht auf Arrays angewendet werden kann, muss allerdings auch im Klassenmodul für jedes Control eine eigene Variable angelegt sowie der VBA-übliche Ereignis-Prozedurcode einzeln notiert werden:

```
'**************MouseDown********************** 'Section02 bis ctlList03 hier nicht dargestellt:
Private Sub Section01_MouseDown(Button As Integer, Shift As Integer, x As Single, Y As Single)
    Standard_MouseDown Section01, Button, Shift, x, Y
End Sub

'**************MouseMove********************** 'Section02 bis ctlList03 hier nicht dargestellt:
Private Sub Section01_MouseMove(Button As Integer, Shift As Integer, x As Single, Y As Single)
    Standard_MouseMove Section01, Button, Shift, x, Y
End Sub

'**************MouseUp********************** 'Section02 bis ctlList03 hier nicht dargestellt:
Private Sub Section01_MouseUp(Button As Integer, Shift As Integer, x As Single, Y As Single)
    Standard_MouseUp Section01, Button, Shift, x, Y
End Sub
```

Die in den `Standard_MouseXXX`-Prozeduren aufgerufenen Prozeduren wie `MouseMoveErweitert` oder `MausPositionSchubsen` werden, soweit nötig, später einzeln betrachtet.

Diese Technik führt dazu, dass die gleichen Ereignisse zur Verfügung stehen, wie sie auch im Formular selber durch umfangreiche Fleißarbeit geschrieben werden können. Tatsächlich ist der Schreibaufwand durch die notwendigen Zuweisungen für die Controls einmalig sogar merklich gestiegen. Aber die Klasse ist neutral formuliert und für viele Formulare gleichzeitig nutzbar, da sie nichts über das konkrete Formular oder Control "weiß".

Vorteile	Nachteile
Extrem geringer Code im Formular, zentraler Code in Klasse mit sehr guter Weiterverwendbarkeit	Controls müssen einzelne Variablen nutzen, daher keine Schleife mit automatischer Erweiterung der Anzahl möglich

Benötigte API-Funktionen

Zwar benutzt auch das SubClassing nur die Windows-eigenen APIs, aber dort greifen sie sehr tief in das zugrundeliegende Nachrichtensystem von Windows ein. Grundsätzlich führt der Einsatz von APIs nicht zum Absturz von Access und kann/muss also als sinnvolle Alternative oder Erweiterung betrachtet werden. Hier geht es daher gezielt darum, die offenen Probleme zu lösen, insbesondere die Tatsache, dass `MouseMove` nicht wirklich das unter der Maus liegende Control erkennt oder dass die Markierung in einer Listbox sich während des Verschiebens ändert.

Die in der Beispieldatenbank vorhandenen drei Formulare *frmDnD_03_clsDD01* bis *frmDnD_05_clsDD03* unterscheiden sich nur darin, dass sie die jeweils weiter optimierten Klassen *clsDragnDrop01* bis *clsDragnDrop03* nutzen.

Leider handelt man sich damit ein ganzes Bündel neuer Probleme ein: Access und API-Funktionen benutzen nicht nur unterschiedliche Koordinatensystem und Einheiten, sondern praktisch allen Access-Controls fehlt es an grundlegendsten Eigenschaften wie einem Windows-Handle, um mit API-Funktionen zusammenzuarbeiten. Schauen wir einfach mal, wie weit das geht.

Positionen in Pixeln

Zuerst soll die Position der Maus via API erfragt werden, damit dies nicht nur unabhängig von den VBA-Ereignissen, sondern vor allem nicht mehr relativ auf das erzeugende Control bezogen ist. Dazu braucht es `GetCursorPos` sowie den zugehörigen Datentyp `POINT`.

Zusätzlich gibt es noch ein paar Umrechnungsfunktionen zwischen Pixeln und Twips, da die API-Funktionen ihre Koordinaten in Pixeln und die Access-Controls ihre in Twips liefern. Auch die von Access ermittelten Angaben zum Formularfenster sind nicht geeignet, weil sie nicht dessen Position innerhalb des Bildschirms berücksichtigen und daher durch die `GetWindowRect`-API-Funktion ersetzt werden müssen.

Vorteile	Nachteile
Mausposition korrekt ermittelbar, Ursprung immer links oben am Bildschirm absolute Koordinaten des Bildschirms	Koordinaten in Pixeln statt Twips

Control unter der Maus

Mit diesen Informationen lässt sich eine Funktion `ObjektUnterMaus` schreiben, welche das im Formular unter der Maus liegende Control zurückgibt. Dies funktioniert auch dann, wenn die Maustaste dabei gedrückt bleibt oder bei Endlosformularen der Fokus in einem anderen Datensatz liegt.

Ein besonderes Problem in Access sind dabei Endlosformulare, bei denen die Koordinaten des Controls um die entsprechende Anzahl Zeilen verschoben werden müssen. Hier greife ich auf die Berechnungen von Stephen Lebans (Quellen siehe weiter unten) als Klasse `clsLebans` zurück:

```
Function ObjektUnterMaus(frmMe As Object) As Control
    Dim pntMaus As POINT
    Dim rctFenster As RECT
    Dim rctX As RECT
    Dim ctlX As Control
    Dim intX As Integer
    Dim intZeile As Integer

    Call GetCursorPos(pntMaus)
    Call GetWindowRect(m_frmDieses.hwnd, rctFenster)
    'Mauskoordinaten kriegen jetzt Ursprung ab Fenster oben links:
    pntMaus.x = pntMaus.x - rctFenster.Left
    pntMaus.Y = pntMaus.Y - rctFenster.Top

    intZeile = clsLebans.WievielteZeile(m_frmDieses) - 1    'also nullbasiert zum Multiplizieren

    'jetzt das Control finden
    For Each ctlX In m_frmDieses.Controls
        rctX.Left = ctlX.Left + m_frmDieses.CurrentSectionLeft
        rctX.Right = rctX.Left + ctlX.Width
        rctX.Top = ctlX.Top
        If ctlX.Section = acDetail Or ctlX.Section = acFooter Or ctlX.Section = acPageFooter Then
            rctX.Top = ctlX.Top + m_frmDieses.CurrentSectionTop
        End If
        'für Endlosformular vorherige Detailbereich-Anzahl und -Höhe draufaddieren
        rctX.Top = rctX.Top + intZeile * m_frmDieses.Section(acDetail).Height
```

```
        If ctlX.Section = acFooter Or ctlX.Section = acPageFooter Then
            'Achtung: Detail-Höhe ist Entwurfshöhe und nicht tatsächliche Darstellung!
            rctX.Top = rctX.Top + m_frmDieses.InsideHeight - _
                m_frmDieses.Section(acHeader).Height - m_frmDieses.Section(acFooter).Height - _
                m_frmDieses.CurrentSectionTop
        End If
        rctX.Bottom = rctX.Top + ctlX.Height
        RectangleTwips2Pixels rctX  'Rechteck in Pixel umrechnen

        If pntMaus.x >= rctX.Left And pntMaus.x <= rctX.Right And _
            pntMaus.Y >= rctX.Top And pntMaus.Y <= rctX.Bottom Then 'die Maus ist über diesem
            Set ObjektUnterMaus = ctlX
            Exit For
        End If
    Next
End Function
```

Obwohl für jede Maus-Bewegung im ungünstigsten Fall die Positionen aller Access-Controls auf einem Formular geprüft werden muss, ist es nicht zeitkritisch. Es gäbe zwar Windows-APIs, welche die Koordinaten innerhalb eines Rechtecks erkennen würden, aber diese benötigen dafür das Windows-Handle (`hwnd`) des Rechtecks. Dieses Windows-Handle besitzen die meisten Access-Controls jedoch gar nicht.

Vorteile	Nachteile
Access-Control unter der Maus auch mit gedrückter Maustaste immer ermittelbar	Könnte bei sehr vielen Controls auf einem Formular zu langsam sein

Maustaste prüfen

Da eine eventuell gedrückte Maustaste so noch nicht ermittelt wird, geschieht dies unabhängig davon über die API-Funktion `GetAsyncKeyState`. Im Vergleich mit verschiedenen Konstanten kann der Code damit zu jedem Zeitpunkt überprüfen, ob beispielsweise gerade die linke Maustaste gedrückt und also ein Drag'n'Drop gewünscht ist.

In diesem Beispiel wäre eine konkrete Prüfung noch nicht einmal nötig, da ein `MouseDown` im Formular zwingend von einer gedrückten Maustaste abhängig ist. Dabei wird eine Modul-öffentliche Variable `m_intModus` auf `DragModus` (aus der Enumeration `enmModus`) umgestellt, so dass eine weitere Abfrage der Maustaste entfallen kann. Beim Loslassen in `MouseUp` wird die Variable entsprechend zurückgestellt. Hier ließe sich bei Bedarf aber zusätzlich die *Strg*-Taste abfragen, um zwischen Verschieben und Kopieren zu unterscheiden.

Vorteile	Nachteile
Taste kann jederzeit geprüft werden	

API-Funktionen aufrufen

Gegenüber der vorherigen Programmierung ändert sich inhaltich nur die `MouseMove`-Funktion im Klassenmodul *clsDragnDrop03*. Sie untersucht in `MouseMoveErweitert`, über welchem Access-Control sich die Maus befindet und ob der Drag- oder Drop-Modus aktiv ist. Davon abhängig wird entweder nur das Cursor-Icon angepasst oder die Drop-Aktion ausgeführt:

```
Private Sub MouseMoveErweitert()
    Dim pntMaus As POINT
    Dim intButton As Integer
    Dim booShift As Boolean

    If GetAsyncKeyState(vbKeyLButton) <> 0 Then intButton = acLeftButton
    If GetAsyncKeyState(vbKeyMButton) <> 0 Then intButton = acMiddleButton
    If GetAsyncKeyState(vbKeyRButton) <> 0 Then intButton = acRightButton
    booShift = (GetAsyncKeyState(vbKeyShift) <> 0)

    Call GetCursorPos(pntMaus)
    Set m_objMove = ObjektUnterMaus()
```

```
    If Not m_objMove Is Nothing Then
        Select Case m_intModus
        Case enmModus.keinModus
            MausFokusHier m_frmDieses.hwnd
        Case enmModus.DragModus
            MausFokusWoanders
            If m_objMove.Tag = "Drop" Then
                LadeCursorDrop
            Else
                LadeCursorNoDrop
            End If

        Case enmModus.DropModus
            MausFokusHier m_frmDieses.hwnd
            Set m_objUp = m_objMove
            ZeigeInfo m_objDown, m_objMove, m_objUp, m_intModus, intButton, booShift, _
                pntMaus.x, pntMaus.Y
            DropMeldung m_objDown, m_objUp
            m_intModus = keinModus
            LadeCursorDefault
            Set m_objDown = Nothing
            Set m_objUp = Nothing
        End Select
    End If

    ZeigeInfo m_objDown, m_objMove, m_objUp, m_intModus, intButton, booShift, pntMaus.x, pntMaus.Y
End Sub
```

Diese Lösung ist stabil und erkennt alle Access-Controls auch bei gedrückter Maustaste rechtzeitig, schnell und eindeutig. Die Programmierung ist auf einige wenige API-Funktionen begrenzt und kann zentralisiert werden.

Vorteile	Nachteile
Die Lösung erkennt jederzeit alle Controls unter der Maus	Einige wenige API-Funktionen notwendig

Listbox-Markierung nicht verschieben

Nicht nur funktionell falsch, sondern auch aus Benutzersicht mehr als irritierend ist die Tatsache, dass beim Verschieben der Maus mit gedrückter Maustaste in Listboxen weiterhin die Markierung geändert wird. Auch das lässt sich nur durch API-Eingriffe in das Maus-Verhalten verhindern.

Mittels `ReleaseCapture` wird der Fokus der Maus während des Verschiebens vom Formular entfernt, so dass auch die Listbox keine weiteren Positionsänderungen des Mauszeiges empfängt. Trotzdem kann die Maus weiterhin die darunterliegenden Access-Controls ermitteln, weil sie ja nur deren Positionen vergleicht.

```
Sub MausFokusWoanders()
    If p_booMausAktiv Then
        Call ReleaseCapture
        p_booMausAktiv = False
    End If
End Sub

Sub MausFokusHier(lngHwnd As Long)
    If Not p_booMausAktiv Then
        Call SetCapture(lngHwnd)
        SetFocusWnd GetCapture()
        p_booMausAktiv = True
    End If
End Sub
```

Im `MouseUp` wird mit `MausFokusHier` der Fokus wieder zum Access-Formular zurückgeholt, damit die nächsten Mauszugriffe wieder funktionieren. Dabei ist es wichtig, mittels einer internen Status-Variablen dieses Fokus-Umschalten möglichst selten stattfinden zu lassen, weil der Maus-Cursor sonst stark flackert.

Vorteile	Nachteile
Listbox-Markierung wird nicht nachträglich verändert	Manchmal "verschwindet" der Mauszeiger

MouseMove nach MouseUp auslösen

Da nach dem `MouseUp` die eigentliche Drop-Aktion erst im nächsten `MouseMove` stattfindet, ist es für den/die Benutzer/in wenig intuitiv, wenn er/sie anschließend erst die Maus bewegen müsste. Das lässt sich leicht per API erledigen, indem die Mausposition direkt um 1 Pixel verändert wird:

```
Sub MausPositionSchubsen()
    Dim pntMaus As POINT

    Call GetCursorPos(pntMaus)
    Call SetCursorPos(pntMaus.X + 1, pntMaus.Y + 1)
End Sub
```

Diese Prozedur wird im `Standard_MouseUp` der Klasse aufgerufen und löst das nächste `MouseMove` auch dann aus, wenn der/die Benutzer/in die Maus noch nicht bewegt hat.

Vorteile	Nachteile
`MouseMove` nach **`MouseUp`** wird automatisch ausgelöst	

Zusammenfassung

Mit API-Funktionen lassen sich die gewünschten Maus-Ereignisse, die Maus-Position und das darunter liegende Objekt sehr zuverlässig erkennen. Zusätzlich kann das nachträgliche Verschieben der Listbox-Auswahl unterbunden werden. In Textboxen bleiben die Markierung und das folgende Verschieben weiterhin sehr mühsam und schwierig. Das ist alles zwar technisch irgendwie korrekt, aber wirklich elegant fühlt es sich nicht an.

Vorteile	Nachteile
Access-Controls werden zuverlässig erkannt Listbox-Markierung wird nicht nachträglich verändert Drag'n'Drop ist möglich	Listbox-Markierung ist etwas schwierig, manchmal "verschwindet" der Mauszeiger. Textboxen sind mühsam zu bedienen

Kein Access

Was also nun? Wer jetzt neidisch auf Word & Co. schaut, bei deren Forms-2.0-Objektmodell das Drag'n'Drop schon fertig eingebaut ist, macht einfach das Nächstliegende, nämlich genau deren Controls auf einem Access-Formular einzusetzen!

Es gibt ein paar Einschränkungen beim Einsatz der Forms-2.0-Controls, die aber nur den/die Entwickler/in treffen: Die Controls müssen immer neu mit dem ActiveX-Steuerelemente-Befehl eingefügt werden und können nicht auf dem Formularentwurf kopiert werden. Außerdem werden manche Eigenschaften wie Schriftart und Schriftgröße nicht im Eigenschaften-Fenster angezeigt, sondern lassen sich nur via Code einstellen.

Und natürlich sind insbesondere Listboxen keine datengebundenen Controls, welche direkt auf Tabellen oder Abfragen zugreifen könnten. Daher müssen diese mit VBA-Recordsets und `AddItem`-Methoden befüllt werden. Der (leicht gekürzte) Code für das entsprechende Beispiel sieht so aus:

```
Private Sub lstFahrrad_MouseMove(ByVal Button As Integer, ByVal Shift As Integer, ByVal X As Single,
ByVal Y As Single)
    Dim dabClipboard As DataObject

    If Button = 1 Then
        Set dabClipboard = New DataObject
        dabClipboard.SetText Me.lstFahrrad.Value
        Call dabClipboard.StartDrag
    End If
End Sub

Private Sub lstZiel_BeforeDragOver(ByVal Cancel As Object, ByVal Data As Object, ByVal X As Single,
ByVal Y As Single, ByVal DragState As Long, ByVal Effect As Object, ByVal Shift As Integer)
    Cancel = True
    Effect = wrdDropEffectCopy
End Sub

Private Sub lstZiel_BeforeDropOrPaste(ByVal Cancel As Object, ByVal Action As Long, ByVal Data As
Object, ByVal X As Single, ByVal Y As Single, ByVal Effect As Object, ByVal Shift As Integer)
    Cancel = True
    Effect = wrdDropEffectCopy
    Me.lstZiel.AddItem Data.GetText
End Sub

Private Sub edtZiel_BeforeDragOver(ByVal Cancel As Object, ByVal Data As Object, ByVal X As Single,
ByVal Y As Single, ByVal DragState As Long, ByVal Effect As Object, ByVal Shift As Integer)
    Cancel = True
    Effect = 1
End Sub
```

```
Private Sub edtZiel_BeforeDropOrPaste(ByVal Cancel As Object, ByVal Action As Long, ByVal Data As
Object, ByVal X As Single, ByVal Y As Single, ByVal Effect As Object, ByVal Shift As Integer)
    Cancel = True
    Effect = wrdDropEffectCopy
    Me.edtZiel.Value = Me.edtZiel.Value & " " & Data.GetText
End Sub
```

Im obigen Code-Ausschnitt ist zu sehen, dass die Start-Controls lediglich ein `MouseMove`-Ereignis und die Ziel-Controls die `BeforeDropOrPaste` bzw. –`BeforeDragOver`-Ereignisse benötigen. Dabei wird durch Rückgabeparameter auch schon der passende Maus-Cursor angezeigt.

Die tatsächliche Datenübergabe aus der Zwischenablage verursacht übrigens so wenig Aufwand, dass dies hier (in kursiv und Kommentar-grün gekennzeichnet) bereits eingebaut ist. Auch der in Access fehlende Zugriff auf das Clipboard-Objekt, welches hier `Data` heißt, wird mit dem Verweis auf Forms 2.0 bereits automatisch mitgeliefert.

Zusammenfassung

So sollte es sein: 3-6 Zeilen je Control reichen komplett aus, um eine perfekte Drag'n'Drop-Funktionalität zu erhalten. Dafür habe ich in der `clsDragnDrop`-Klasse plus diversen Hilfsprozeduren über 400 Zeilen Code gebraucht und trotzdem noch Probleme und Unzulänglichkeiten gefunden!

Ohne Änderung funktioniert die Forms-2.0-Version auch direkt in Unterformularen oder zwischen mehreren Formularen. Lediglich von einem Datensatz im Endlosformular zu einem anderen geht es nicht.

Vorteile	Nachteile
Sehr wenige Zeilen VBA-Code, intuitive Markierung auch für Textboxen, auch für Unterformulare und zwischen Formularen	Keine direkte Datenbindung, manche Eigenschaften nur via VBA einstellbar, nicht zwischen verschiedenen Datensätzen im Endlosformular

Dateinamen vom Explorer nach Access ziehen

Eine besondere Version von Drag'n'Drop ist das Hineinziehen von Informationen aus einem anderen Programm nach Access. Dabei gibt es drei Herausforderungen:

- Es können beliebige Objekte (Dateien, Bildausschnitte, Texte, Tabellen, Vektorgrafiken, etc.) gezogen werden, je nach ursprünglichem Programm.
- Access muss "erfahren", dass eine Drop-Aktion stattfindet.
- Die Dateinamen müssen aus der Zwischenablage extrahiert werden.

Da Access typischerweise Daten verarbeitet, verzichte ich auf eher exotische Formate wie Bildausschnitte oder Vektorgrafiken. Der häufigste Wunsch nach Drag'n'Drop besteht sicherlich in der Auswahl von Dateinamen aus dem Explorer, was hier exemplarisch untersucht werden soll.

Die eigentlichen Schwierigkeiten liegen sowieso im zweiten und dritten Punkt: Access muss eine Nachricht über das Loslassen der Maus über seinem Fenster erhalten und daraufhin eine Prozedur starten, welche die Dateinamen aus der Zwischenablage ausliest.

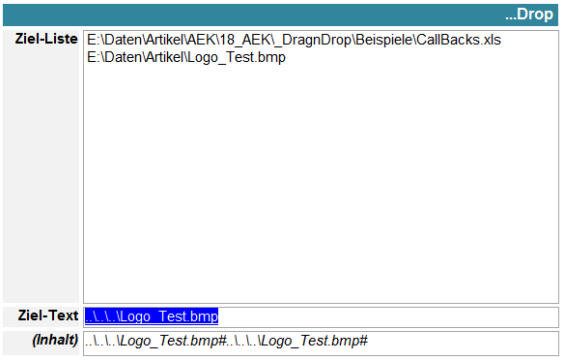
Es gibt allerdings fast kein Access-eigenes Control, welches auf ein Drop-Ereignis von außen reagiert. Die einzige Chance bestünde darin, mit SubClassing diese Windows-Message abzufangen, was allerdings absturzträchtig ist oder wieder eine externe DLL erzwingt.

Die Alternative mit den Forms-2.0-Controls hilft leider auch nicht weiter, denn dort wird zwar das Drop-Ereignis erkannt und dabei auch via `Data`-Objekt der Zugriff auf die Zwischenablage erlaubt. Allerdings ist dieses Objekt nur in der Lage, reine Texte aus der Zwischenablage zu lesen. Das sogar bereits vordefinierte Clipboard-Format für Dateien/Dateinamen wird als ungültig verweigert.

Interessanterweise gibt es aber doch ein Control in Access, welches manchmal bereits komplett ohne VBA-Programmierung das Drag'n'Drop beherrscht:

- Textboxen, deren Textformat auf "Rich-Text" eingestellt ist, beherrschen das Drag'n'Drop, so dass sich Textteile darin markieren und verschieben lassen. Das funktioniert sowohl innerhalb der gleichen Textbox als auch zwischen verschiedenen. Allerdings wird beim Verschieben zu einer anderen Textbox deren vorheriger Inhalt ohne Warnung komplett gelöscht.
- Sobald eine gebundene Textbox den Inhalt eines Link-Felds darstellt, lassen sich direkt Dateien aus dem Explorer hineinziehen. Als Inhalt wird der Dateiname angezeigt.

Damit ist die wesentliche Hürde genommen, insbesondere im zweiten Fall des Hereinziehens von Dateinamen. Allerdings enthält dieser Dateiname einen relativen Pfad, wie in der folgenden Abbildung zu sehen ist, und Dateien aus dem gleichen Verzeichnis enthalten überhaupt keinen Pfad. Daher gibt es hier eine zusätzliche Funktion, welche diese relativen Pfade in absolute umsetzt und das Ergebnis oben in die Listbox schreibt.



Durch Ziehen eines Dateinamens aus dem Explorer in die Ziel-Textbox wird eine Prozedur gestartet, welche aus dem tatsächlichen Inhalt (siehe Zeile darunter) einen kompletten gültigen Pfad erzeugt und in der Listbox einträgt.

Ohne weitere Fehlermeldung wird übrigens auch bei mehreren gezogenen Dateinamen immer nur ein einziger in das Feld aufgenommen.

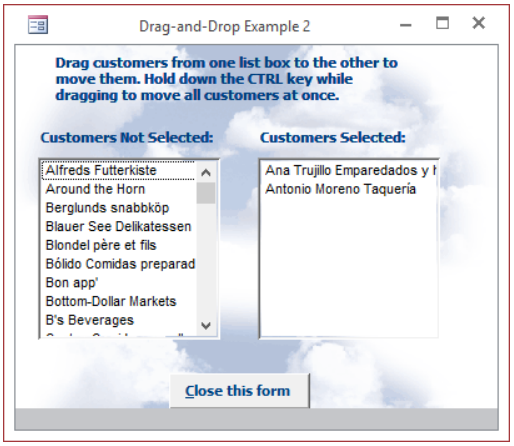
Vorteile	Nachteile
Access-interne Programmierung, keine DLL nötig, minimaler Code-Aufwand	Jeweils nur ein Dateiname, Pfade müssen umgewandelt werden, Datenbindung an Tabellenfeld notwendig

Beispiel-Datenbanken

Ich bin nicht der Erste (und bestimmt nicht der Letzte), der sich an diesem Thema versucht. Es gibt bereits verschiedene Teil-Lösungen, auf die ich hier hinweisen will. Diese Auflistung ist nicht vollständig, sondern soll lediglich Anregungen liefern. Und natürlich werde ich auch die hier gezeigte Lösung als Download zur Verfügung stellen, so dass Ihr diese nutzen könnt.

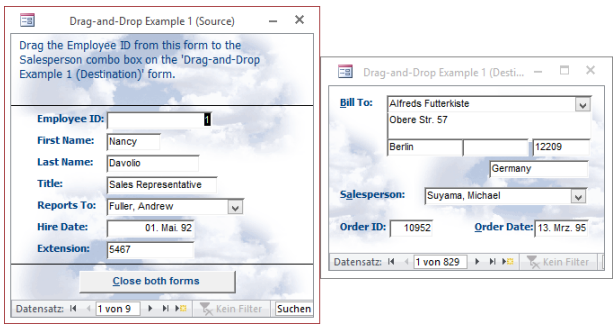
Microsoft-Beispieldatenbank

Unter https://www.microsoft.com/en-us/download/details.aspx?id=12100 bietet Microsoft eine Access2000-mdb an, in welcher zwei Demos für Drag'n'Drop enthalten sind. Das eine zeigt, wie Werte aus einer Listbox in eine zweite Listbox auf dem gleichen Formular gezogen werden können.



Per Maus lassen sich Werte aus der linken Listbox nach rechts ziehen.

Das zweite Demo zeigt das Drag'n'Drop zwischen zwei einzelnen Formularen:



Der Benutzer kann hier die *Employee-ID* aus dem linken Formular markieren und nach rechts auf *Salesperson* ziehen, damit deren Eintrag dort wechselt

Beide Demos benutzen die üblichen VBA-Ereignis-Prozeduren und rufen die gemeinsamen zentralen Prozeduren `DragStart`, `DragStop`, `DragDetect` und `DragDrop` auf. An diesen oder sehr ähnlichen Prozedurnamen sind auch die im Internet weit verbreiteten Kopien dieses Demos zu erkennen.

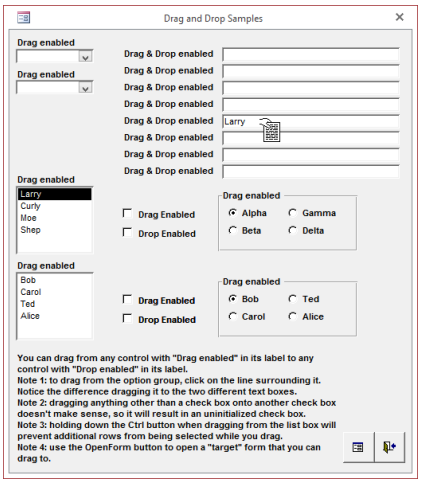
Inhaltlich prüft das Ziel-Control in seinem `MouseMove` mittels `DropDetect`, ob eine öffentliche Variable auf `DROP_MODE` steht und außerdem eine definierte Zeitspanne nicht überschritten wurde. Dann wird `DragDrop` aufgerufen, in welchem hart codiert der Name des einzig erlaubten Start-Formulars geprüft und anschließend ein ebenfalls hart codiertes Feld in einer festgelegten Tabelle ausgelesen wird.

Diese Lösung ist also unflexibel und legt sich beim Drop auf exakte Controls und Datenquellen fest. Für jede Listbox muss eine einzelne (Teil-)Lösung hinzuprogrammiert werden. Im Übrigen gibt es im Maus-Cursor keine Rückmeldung an den Benutzer, weder beim Ziehen noch beim Überfahren eines optionalen Ziel-Controls. Das ist, gelinde gesagt, überhaupt nicht intuitiv.

Doug Steele

Von Doug Steele gibt es eine Beispiel-Datenbank für Drag'n'Drop, deren Code teilweise identisch ist mit demjenigen des MS-Demos, allerdings bedeutend umfangreicher kommentiert. Der Download der eigentlichen Datenbank ist etwas schwierig zu finden, es gibt einen Link in einem Beitrag des britischen Access-Programmers-Forums (http://www.access-programmers.co.uk/forums/showthread.php?p=1130132) sowie eine ausführliche Erläuterung auch unter http://www.vb123.com/Pages/kb/201202_ds_drag.aspx.

Verschiedene (inhaltlich aber sehr ähnliche) Formularfenster zeigen, wie Daten zwischen Controls per Drag'n'Drop gezogen werden können. Im folgenden Bild ist sozusagen der Maximal-Dialog mit allen Beispielen zu sehen. Technisch werden wie bei MS die zentralen Prozeduren `StartDrag`, `StopDrag`, `DetectDrop` und `ProcessDrop` aufgerufen und darin hart codierte Controls mit fixierten SQL-Anweisungen befüllt.



In verschiedenen Dialogen (hier der umfangreichste) lassen sich Inhalte zwischen verschiedenen Controls hin und her ziehen. Der Mauszeiger hat hier gerade den ersten Wert der Listbox in die fünfte Zeile gezogen.

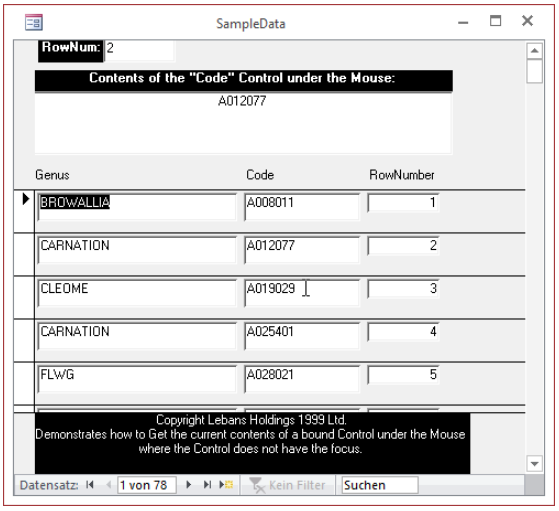
Dabei gibt es erwartungsgemäß die gleichen Probleme, dass der Mauszeiger beim Ziehen nicht auf das potentielle Ziel-Control reagieren kann und in Listboxen bei senkrechter Mausbewegung fälschlich nachträglich weitere Listenelemente ausgewählt werden.

Stephen Lebans (+ Dev Ashish)

Auf http://www.lebans.com/conformscurcontrol.htm zeigt Stephen Lebans (teilweise basierend auf Code von Dev Ashish), wie es möglich ist, auf die unter der Maus liegenden Inhalte eines Endlosformulars zuzugreifen. Es handelt sich dabei allerdings um eine 97er-mdb, die zuerst in ein accdb-Format umgewandelt werden muss, damit sie heute ausführbar ist.

Dies ist eher ein Proof of Concept und keine sofort einsetzbare Lösung für alle Fälle. Hier wird dafür sehr sauber die tatsächliche Position der Access-Controls berechnet, die nicht nur vom n-ten Datensatz im Endlosformular abhängig sind, sondern außerdem noch von der Höhe der Ribbons und der Fensterrahmen usw. Allerdings erhält die aufrufende Prozedur nicht das Objekt zurück, sondern das Modul schreibt dessen Inhalt in eine anzugebende Textbox.

Abgesehen davon, dass der Code sich dabei leider um exakt einen Datensatz verrechnet, reagiert er auch nur auf ein einziges Zielfeld (hier `Code`), dessen Inhalt ausgelesen wird. Tatsächlich werden die Daten gar nicht aus dem Steuerelement selber gelesen, sondern anhand der `RowNum` zum entsprechenden Datensatz der zugrundeliegenden Datenquelle gesprungen. Trotzdem ist die Berechnung der Positionen und der zugehörigen Endlos-Zeile sehr stabil.

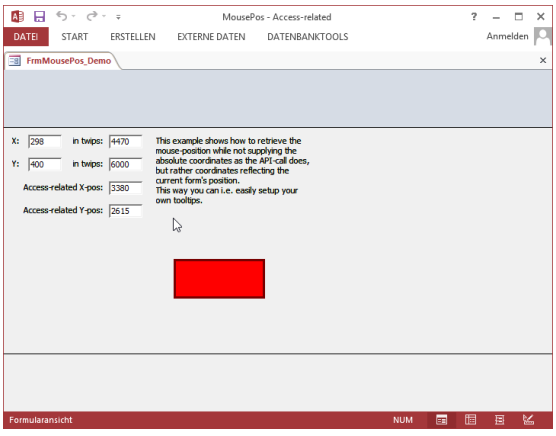


Der Inhalt und die Zeilennummer unter der Maus werden im oberen Teil des Formulars angezeigt. Da allerdings die erste Zeile als 0 gezählt wird, zeigt der Formularkopf immer den Wert darüber an.

Ich habe daher dieses Modul in mein Beispiel übernommen und um drei Funktionen erweitert, welche ein paar der intern berechneten Positionen von außen nutzbar machen, so dass damit das Objekt unter der Maus zuverlässig ermittelbar ist.

Olaf Rabbachin

Auch Olaf Rabbachin stellt auf http://www.blogs.intuidev.com/post/2002/01/01/AccessMousePos.aspx ein Proof Of Concept vor, wie die Maus-Position ermittelt werden kann. Es handelt sich zwar um eine mdb, sie ist aber auch unter heutigen Access-Versionen lauffähig.



Das rote Viereck folgt dem Mauszeiger innerhalb der Grenzen des Detailbereichs. Parallel dazu werden in den Textboxen die jeweiligen Maus-Koordinaten angezeigt.

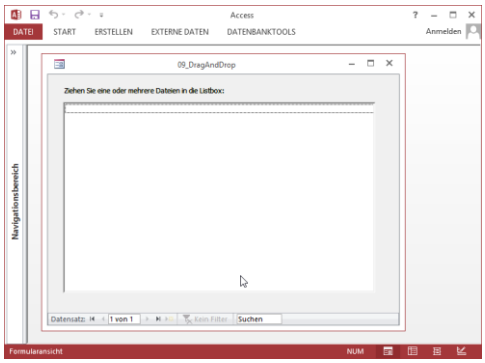
Ein Formular zeigt laufend die Koordinaten des Mauszeigers in Pixeln und Twips an. Ein rotes Rechteck folgt der Bewegung und ist dabei gezielt auf die Grenzen des Detailbereichs beschränkt. Werden diese überschritten, erscheint eine automatische Warnung.

Auch hier werden die Koordinaten inklusive Fensterrahmen und unter Berücksichtigung der Bildschirmauflösung ermittelt. Allerdings wird die Höhe der heute meist üblichen Registerkarten über dem Access-Formularfenster noch nicht berücksichtigt und die Lösung funktioniert nicht mit Endlosformularen.

Jörg Ostendorp (+ Dev Ashish), AEK 10

Ein ganz anderes Beispiel findet sich im Download zur AEK 10 (http://www.donkarl.com/Downloads/AEK/) von Jörg Ostendorp. Dort geht es um die Möglichkeit, Dateinamen durch Drag'n'Drop aus dem Explorer in eine Access-Listbox zu ziehen. Zu der Access2000-mdb braucht es daher die *SSubTmr6*-DLL, die in Access als Verweis eingebunden werden muss.

Die VB-DLL registriert ein File-Drop durch Überwachung der `WM_DROPFILES`-Message. Mit der Funktion `DragQueryFile` können die Anzahl und Namen der Dateien ermittelt und als `RowSource`-String an die Listbox übergeben werden.

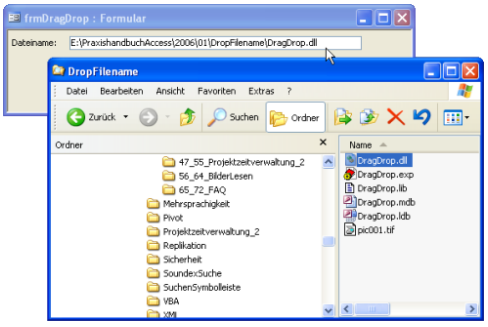


In diese Listbox werden die Dateien aus dem Explorer gezogen, um dort dann die Dateinamen aufzulisten.

Leider lässt sich zwar die DLL registieren und die Datenbank starten, das Hineinziehen der Dateien führt aber zuverlässig zu einem Absturz (Access 2013 unter Win 8.1). Zudem verbieten viele Kunden den Einsatz einer fremden DLL auf ihren Rechnern, so dass eine solche Lösung dann ohnehin nicht möglich wäre.

André Minhorst, Access im Unternehmen

André Minhorst zeigt in einem Artikel seiner Zeitschrift *Access im Unternehmen*, wie Drag'n'Drop für Dateinamen nachgerüstet werden kann (http://www.access-im-unternehmen.de/index1.php?id=300&BeitragID=326). Dazu ist ebenfalls zwingend eine DLL notwendig, deren VB-Code für VisualStudio 6.0 dort dokumentiert ist bzw. als Download bereitsteht.



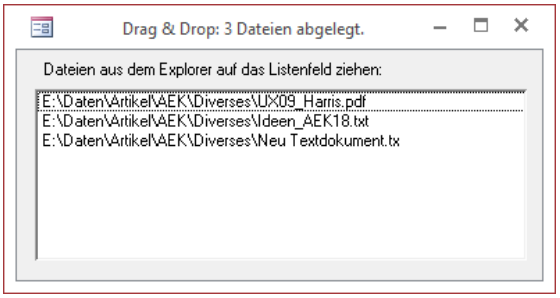
Der Dateiname aus dem Explorer lässt sich in das Access-Formular hineinziehen.

Die DLL enthält zwar nur wenigen und recht übersichtlichen Code mit den üblichen API-Funktionen wie `DragQueryFile` oder `SetWindowLong`, aber auch hier bleibt das grundsätzliche (und nicht vermeidbare) Problem einer externen DLL bestehen.

Reinhard Kraasch, dbwiki

Eine Access2000-mdb von Reinhard Kraasch (http://www.dbwiki.net/wiki/Access_Beispieldatenbanken) findet sich im dbwiki als Download, bei welcher Dateien aus dem Explorer in ein Access-Listenfeld gezogen werden können. Dort werden dann deren Pfade und Namen angezeigt.

Diese Lösung kommt ohne externe DLL aus, sondern hängt sich via SubClassing in das Message-System des Formulars, um dort die Nachricht `WM_DROPFILES` zu erkennen. Dann werden eine Reihe von API-Aufrufen genutzt, um die Namen der in der Zwischenablage enthaltenen Dateien auszulesen und in die Listbox zu schreiben.



Mehrere Dateien wurden aus dem Explorer in das Listenfeld gezogen und zeigen dort ihre Pfade und Dateinamen an.

Die Datenbank funktioniert (getestet für Access 2013/Win 8.1) einwandfrei, solange der VBA-Editor nicht geöffnet ist. Ein Wechsel zu diesem zeigt zuerst ein starkes Flackern seiner Titelleiste und lastet dann mindestens einen Prozessor so vollständig aus, dass Windows mehrere Minuten braucht, um überhaupt wieder zu reagieren. Das ist kein kompletter Absturz, aber kein arbeitsfähiger Zustand mehr und lässt sich nur durch Neustart von Access beseitigen.

Zusammenfassung

Drag'n'Drop macht Spaß, aber nur für BenutzerInnen. Für Access-EntwicklerInnen bleibt offenbar nur die Auswahl zwischen Pest, Cholera und Pocken:

- Eine externe DLL, welche das SubClassing nach außen verlagert und dadurch Abstürze vermeidet. Das benötigt eine zweite Programmiersprache und die Installation der DLL beim Kunden.
- Eine Drag'n'Drop-Klasse, welche die Events kapselt und dafür in Formularen einfach aufzurufen ist. Das führt zu einigen Hundert Zeilen Code und trotzdem zu kleinen Rest-Problemen bei der Erkennung aller Controls.
- Die Nutzung von Forms-2.0-Controls, welche die Drag'n'Drop-Funktionalität von Haus aus mitbringen. Sie können keine Access-Datenquellen nutzen, sondern müssen via VBA-Recordset befüllt werden.

Ich persönlich bevorzuge für das "normale" Drag'n'Drop die letzte Variante, weil sie zuverlässig ist und mit extrem wenig Code auskommt. Für das Drag'n'Drop von Dateien aus dem Explorer ohne DLL kommt nur eine RichText-Textbox mit Tabellen-Datengrundlage in Frage, auch wenn diese jeweils höchstens einen einzigen Dateinamen annimmt. Immerhin.

Die Beispiel-Datenbank steht als Download wie immer allen AEK-TeilnehmerInnen für Spiel, Spaß und Spannung zur Verfügung.

Lorenz Hölscher

Nachtrag

Ich habe nach den Vorträgen von Kollegen noch Hinweise auf Verbesserungsmöglichkeiten erhalten, die ich Euch nicht vorenthalten will. Die Datenbank selber ist sowieso aktualisiert und hier folgen die Erläuterungen.

AccHitTest

Von Karl Donaubauer bin ich an die `AccHitTest(ptX, ptY)`-Funktion erinnert worden. Diese undokumentierte Funktion ermittelt anhand der Maus-Koordinaten (interessanterweise in Bildschirm-Pixeln und nicht in den üblichen Access-Twips…), welches Access-Objekt sich unter der Maus befindet. Diese Funktion erspart also das langwierige Suchen in allen Objekten und sie funktioniert erfreulicherweise auch in Endlos-Formularen.

Das ist ein aussichtsreicher Kandidat, um das Hauptproblem zu lösen. Die Gelegenheit ist nebenbei günstig, um überhaupt auf die in den MouseXXX-Ereignissen falsch oder gar nicht übergebenen Objekte zu verzichten. In allen drei Ereignissen wird das Objekt mit `AccHitTest` ermittelt und damit kann ich extrem Code-sparend mit VBA-injizierten Functions arbeiten.

Ein allgemeines Modul `modDragnDrop` enthält die allgemeinen Prozeduren sowie die Technik zum Injizieren in die Formular-Controls:

```
Dim m_frmMe As Form
Dim m_ctlDown As Control
Dim m_ctlMove As Control
Dim m_ctlUp As Control
Dim m_intModus As enmModus
Dim m_pntMaus As POINT


Sub VerbindeDragnDrop(frmMe As Form, strListeControls As String)
    Dim ctlX As Control

    If p_clsEDD Is Nothing Then
        Set p_clsEDD = New clsEventsDragnDrop
    End If

    Set m_frmMe = frmMe
    On Error Resume Next    'wegen Labels o.ä.
    For Each ctlX In frmMe.Controls
        If InStr(strListeControls, ctlX.Name) Then
            ctlX.OnMouseDown = "=MouseDown_HitTest()"
            ctlX.OnMouseMove = "=MouseMove_HitTest()"
            ctlX.OnMouseUp = "=MouseUp_HitTest()"
        End If
    Next
    On Error GoTo 0
    m_intModus = keinModus
End Sub

'**********Mouse-Ereignisse*******************
Function MouseDown_HitTest()
    GetCursorPos m_pntMaus
    Set m_ctlDown = ObjektUnterMaus_HitTest()
    Set m_ctlMove = Nothing
    Set m_ctlUp = Nothing
    m_intModus = DragModus

    LadeCursorDrag m_ctlDown.Name
    ZeigeInfo m_ctlDown, m_ctlMove, m_ctlUp, unbekannt, 99, 99, m_pntMaus.X, m_pntMaus.Y
End Function

Function MouseMove_HitTest()
    Dim strCtlName As String

    GetCursorPos m_pntMaus
    Set m_ctlMove = ObjektUnterMaus_HitTest()
'siehe Datenbank
End Function

Function MouseUp_HitTest()
'    MausFokusHier FindeParentMitHwnd(m_ctlMove)

    GetCursorPos m_pntMaus
    Set m_ctlUp = ObjektUnterMaus_HitTest()
'siehe Datenbank
End Function
```

```
Private Function ObjektUnterMaus_HitTest() As Control
    Dim frmX As Form
    Dim ctlX As Control
    Dim strCtlName As String

    GetCursorPos m_pntMaus
    On Error Resume Next
    Set ctlX = Nothing
    Set ctlX = m_frmMe.accHitTest(m_pntMaus.X, m_pntMaus.Y)

    If ctlX Is Nothing Then  'also in anderen Formularen suchen...
        For Each frmX In Forms
            Set ctlX = frmX.accHitTest(m_pntMaus.X, m_pntMaus.Y)
            If Not ctlX Is Nothing Then  'also dort eines gefunden!
                Exit For
            End If
        Next
    End If
    On Error GoTo 0

    Set ObjektUnterMaus_HitTest = ctlX
End Function
```

Die `AccHitTest`-Funktion bezieht sich allerdings auf ihr Eltern-Objekt, also ein bestimmtes Formular. Um ein Drag'n'Drop zwischen Formularen zu erlauben, muss der Code bei Bedarf alle geöffneten Formulare nacheinander durchsuchen, bis eines der Objekte erkannt wird. Das ist aber schnell genug.

Allerdings widerspricht die `AccHitTest`-Funktion dem Einsatz von `ReleaseFocus`, welcher nötig ist, damit sich die Auswahl in einer Listbox beim Drag'n'Drop nicht nachträglich verändert. Wenn der Maus-Fokus dem Formular entzogen wird, liefert die Funktion keine Objekte mehr. Die von mir entwickelte Variante, bei der die Maus-Koordinaten mit den Koordinaten aller Objekte auf allen Formularen verglichen werden, funktioniert hingegen weiterhin auch ohne Maus-Fokus.

FilesDropped: UtterAccess

Harald Hattinger hat mich auf einen Forumsbeitrag (http://www.utteraccess.com/forum/index.php?showtopic=1973842) mit einer Beispieldatenbank zum Hereinziehen mehrerer Dateinamen aufmerksam gemacht. Es ist eine 2003er-mdb, die in aktuellen Access-Version geladen werden kann. Der dort enthaltene VBA-Code ist in meiner Datenbank als Modul `modDragDrop_UtterAccess` originalgetreu übernommen.

Das Konzept besteht darin, zwar die absturzgefährdenden Message-Hook-APIs zu benutzen, dieser aber via Timer nur ganz kurzfristig aufzurufen. Dadurch gelingt es offensichtlich, die Vorteile sogar mehrerer gleichzeitig gezogener Dateinamen zu nutzen, ohne die Nachteile der schweren Access-Abstürze zu erleiden.

Endgültige Zusammenfassung

Nach Sichtung der zusätzlichen Möglichkeiten sind meine Favoriten nun so:

- Für das Drag'n'Drop zwischen zwei Listboxen gibt es zwei Lösungen:
  - Mit reinem Access-Bordmitteln die Klasse `clsDragnDrop03_ohneVerschieben`, welche ohne die `AccHitTest`-Funktion, aber dafür unter Beibehaltung der markierten Zeile in der Listbox funktioniert.
  - Beim Vorhandensein der *Forms-2.0*-Library aus den übrigen Office-Programmen lässt sich dort mit weniger als 5 Zeilen Code ein vollständig funktionierendes Drag'n'Drop einschalten. Wenn diese zur Verfügung steht, ist das mit Abstand die einfachste Lösung.
- Für das Drag'n'Drop zwischen Textboxen funktionieren nur die Textboxen aus der *Forms-2.0*-Library, weil die Access-eigenen Textboxen bei jedem Versuch, ein markiertes Textstück zu verschieben, stattdessen sofort dessen Markierung aufheben.
- Für `FilesDropped` das Modul `modDragDrop_UtterAccess`, mit dem sich beliebig viele Dateien aus dem Explorer in eine Textbox ziehen und deren Dateinamen ermitteln lassen.

Die anhand der Rückmeldungen verbesserte Beispiel-Datenbank *DragNDrop.accdb* sowie die (absturzträchtige) *SubClassing.accdb* stehen als Download allen AEK-TeilnehmerInnen zur Verfügung. Dazu kommt ausnahmsweise noch die im Vortrag benutzte PowerPoint-Präsentation, weil dort die Argumente übersichtlich aufgelistet sind.

Lorenz Hölscher