Drag Your Applications into the 21st Century with Drag-and-Drop

Tamar E. Granor
Tomorrow's Solutions, LLC
Voice: 215-635-1958
Website: www.tomorrowssolutionsllc.com
Email: tamar@tomorrowssolutionsllc.com

Users today expect to be able to interact with applications by dragging and dropping. FoxPro has offered drag-and-drop capability since VFP 3, but many developers have never worked with it. In fact, VFP offers two separate approaches for drag-and-drop, a native capability and OLE drag-and-drop, which allows you to interact with other applications and gives you greater control over the process.

In this session, we'll look at both approaches and show both how easy simple drag-and-drop is, and the complex things you can accomplish with it.

Introduction

Drag-and-drop has been part of user interfaces for decades. Visual FoxPro 3 introduced the ability to provide drag-and-drop in VFP applications and VFP 5 added the ability to drag-and-drop between VFP and other applications. Yet most VFP applications I've encountered don't offer drag-and-drop capabilities, even where they would make user's lives easier.

That may be because getting drag-and-drop right can be a little tricky. There are two different drag-and-drop systems built into VFP and each them offers multiple approaches. But with a little effort up front, you can give users the ability to move things around both within VFP applications and between VFP applications and other applications.

Why are there two?

Though FoxPro 2.0 and later included drag-and-drop in the IDE (interactive development environment), the language didn't have support for putting it into your applications. VFP 3, however, included drag-and-drop functionality in the base classes for forms and all the visible controls. For the first time, we could easily create forms that let users move controls around.

However, in Windows (and on Macs), drag-and-drop operates between applications, not just within individual applications. VFP 3 didn't provide a way to do that, presumably because it was a cross-platform product, with versions for Windows, Mac and Unix. (A DOS version was originally announced, but never completed.)

With the commitment to Windows-only in VFP 5, there was no longer a need to ensure drag-and-drop worked across platforms. So, in VFP 6, support was added for cross-application drag-and-drop, known as *OLE drag-and-drop*, but support for native drag-and-drop was retained as well for backward compatibility.

The names of all the properties, events and methods (PEMs) for native drag-and-drop begin with "Drag." The names of the PEMs for OLE drag-and-drop begin with "OLE" (except for those belonging to the data object, explained later in this paper).

This paper covers both approaches, but for the most part, I recommend OLE drag-and-drop. It's more powerful and offers a finer-grained approach.

Basics

Whichever type of drag-and-drop you choose, two objects are involved at any time:

- the *drag source*, the control or application you're dragging from;
- the *drop target*, the control or application onto which data may be dragged.

In OLE drag-and-drop, there's a third object, called the data object, which is an OLE object containing the data from the drag source in multiple formats.

The VFP base classes include PEMs for an object to serve as either a drag source or a drop target. In fact, an object can be both.

Both versions support both manual and automatic dragging. That is, whether you're using native drag-and-drop or OLE drag-and-drop, you can set a control so that clicking on it and starting to move begins a drag automatically, or you can require a method call from MouseDown (or, for native drag-and-drop only, MouseMove) to start dragging. Note that at any given time, a control can be a drag source with either native drag-and-drop or OLE drag-and-drop, but not both.

With native drag-and-drop, there's no property to indicate whether an object is a drop target. You can use code to change the icon to the "no drop" icon when over a control onto which the user shouldn't drop. OLE drag-and-drop gives you more control here; a property indicates whether an object allows drops.

Table 1 shows the PEMs involved in both native and OLE drag-and-drop. As the table shows, OLE drag-and-drop provides a lot more control and, in fact, there are additional methods that belong to the data object. The details of all these PEMs are discussed later in this document.

Table 1. Controlling drag-and-drop involves a number of properties and methods.

Purpose	**Native**	**OLE**	**Comments**
Control automatic drag	DragMode	OLEDragMode	Property of drag source: 0=manual 1=automatic
Start manual drag	Drag	OLEDrag	Method of drag source
Respond to start of drag		OLEStartDrag	Event of drag source
Respond to end of drag		OLECompleteDrag	Event of drag source
Control icon seen while dragging	DragIcon	OLEDragPicture	Property of drag source
Show user result of possible drop		OLEGiveFeedback	Event of drag source
Control whether object is drop target		OLEDropMode	Property of drop target
Respond to drag over object	DragOver	OLEDragOver	Event of drop target
Respond to drop onto object	DragDrop	OLEDragDrop	Event of drop target
Indicate whether the drag source has useful data		OLEDropHasData	Property of drop target
Indicate what types of drops are accepted		OLEDropEffects	Property of drop target
Indicates whether text can be inserted in the middle of existing text		OLEDropTextInsertion	Property of drop target: 0=insert anywhere 1=insert at beginning of word. Broken in VFP 7 and later; has no effect.
Specify the actual data to be dropped		OLESetData	Event of drag source

Native Drag-and-drop

VFP's native drag-and-drop lets you drag controls within VFP forms and from one VFP form to another, but only within a single instance of VFP or a VFP executable. You can determine how dragging starts, what shows when you're dragging, what happens when you drag over a control and what happens when you drop.

There are two ways to start a native drag. The first is to set a control's DragMode property to 1-Automatic. When you do that, as soon as you click the mouse down onto the control, dragging begins. Dragging normally ends when you release the mouse button.

The second way to start a native drag is to call the Drag method of a control and pass 1 as a parameter. The usual place to do that is in the MouseDown or MouseMove method of the control. Since you don't necessarily want to start dragging just because the user pushed the button down and then moved a tiny bit, it's common to put code in MouseDown to save the mouse position when the mouse button was clicked and use code in MouseMove to check whether you've moved far enough to trigger dragging, as in **Listing 1**. (This code is included in the most of the drag-and-drop base classes in the session materials. The class library is dragdropbase.vcx.)

Listing 1. Code like this in the MouseMove event lets you start dragging only if the user has moved the mouse enough to warrant it.

```
LPARAMETERS nButton, nShift, nXCoord, nYCoord

IF m.nButton = 1
   IF ABS(m.nXCoord - This.nMouseX) + ABS(m.nYCoord - This.nMouseY) > 5
      This.Drag(1)
   ENDIF
ENDIF
```

Be aware that this approach doesn't work well for all of the base classes. Specifically, those that allow input by typing (textboxes, editboxes, spinners, and comboboxes with Style set to 0-dropdown combo) are hard to drag when using this approach. You have to click over a part of the control that can't accept focus, such as a border or scroll bar to start dragging.

The DragIcon property lets you control what the user sees when dragging. Although the name contains "icon," the property accepts both cursor files (.CUR) and icon files (.ICO). You can specify DragIcon at design-time or at run-time. One common use for this property is to indicate that a particular object doesn't accept drops from the dragged control by setting the drag source's DragIcon to the universal "No" sign (included with VFP as HOME(4) + "Cursors\NoDrop01.CUR") in the drop target's DragOver method.

Two events control what happens when you're dragging, DragOver and DragDrop. The DragOver event of a drop target fires when you drag over it, while DragDrop fires when you release the mouse button while over the object. Both have parameters to indicate the drag source, as well as the mouse position.

DragOver has an additional parameter to indicate where the drag source is with respect to the drop target's space: entering (0), leaving (1), or still within (2). As described above, you might use that to change the drag source's DragIcon, using code like that in **Listing 2**. This code also saves and restores the drag source's original DragIcon, by creating a property of the drag source if necessary.

Listing 2. Use code like this to indicate that an object doesn't accept a drop.

```
LPARAMETERS oSource, nXCoord, nYCoord, nState

DO CASE
CASE m.nState = 0 && Entering, no drops here
   * Save prior icon
   IF PEMSTATUS(oSource, "cDragIcon", 5)
      oSource.cDragIcon = oSource.DragIcon
   ELSE
      oSource.AddProperty("cDragIcon", oSource.DragIcon)
   ENDIF

   oSource.DragIcon = HOME(4) + "Cursors\NoDrop01.CUR"

CASE m.nState = 1 && Leaving, so reset
   * There should always be something to reset it to, but jic
   IF PEMSTATUS(oSource, "cDragIcon", 5)
      oSource.DragIcon = oSource.cDragIcon
   ELSE
      oSource.ResetToDefault("DragIcon")
   ENDIF

OTHERWISE
   * Nothing to do
ENDCASE
```

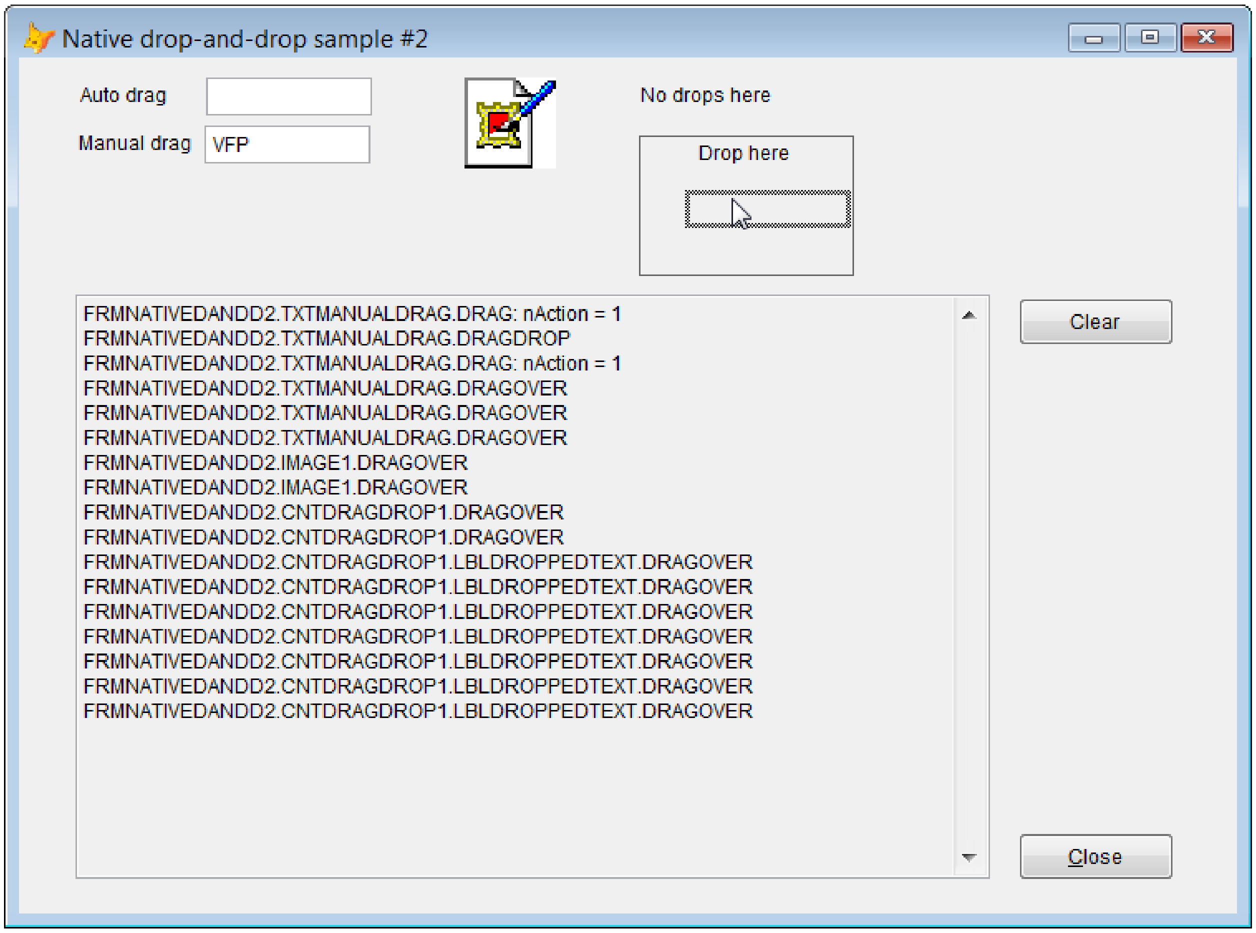
The code can also interrogate the drag source, so that it behaves differently depending on what object or what kind of object is being dragged. So, for example, in a two-column mover (see "A two-column mover," later in this document), each list could indicate that it accepts drops only from the other list, by showing the no drop icon for any other drag source.

Put code to take action on a drop in DragDrop. For example, you could change the text of a label in its DragDrop event with code like **Listing 3**. As the example shows, here, too, you can interrogate the DragSource to determine the action to take.

Listing 3. Use the DragDrop event to respond to drops.

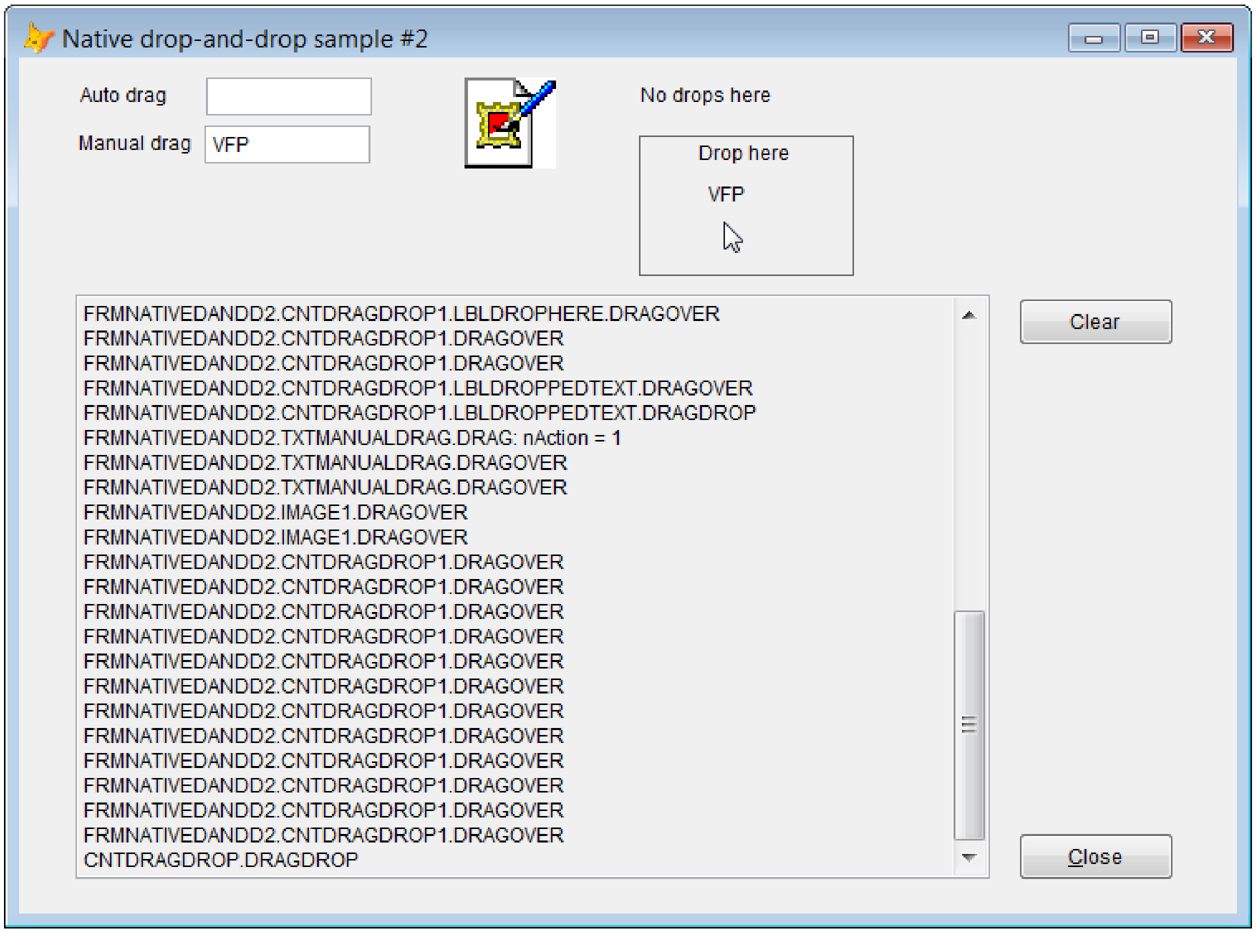
```
LPARAMETERS oSource, nXCoord, nYCoord

IF PEMSTATUS(oSource, "Text", 5)
   This.Caption = oSource.Text
ENDIF
```

Figure 1 shows the form NativeSample2.SCX, included in the downloads for this session, just before dropping the textbox labeled "Manual drag" on the container with the "Drop here" label. **Figure 2** shows the same form after the drop. The container has a second label, whose Caption is initially blank (thus hiding the label); the container's DragDrop has code similar to that in **Listing 3**, but it sets the label's Caption. (This form, like many in this session, is designed to show all drag-and-drop events as they fire.)



```
FRMNATIVEDANDD2.TXTMANUALDRAG.DRAG: nAction = 1
FRMNATIVEDANDD2.TXTMANUALDRAG.DRAGDROP
FRMNATIVEDANDD2.TXTMANUALDRAG.DRAG: nAction = 1
FRMNATIVEDANDD2.TXTMANUALDRAG.DRAGOVER
FRMNATIVEDANDD2.TXTMANUALDRAG.DRAGOVER
FRMNATIVEDANDD2.TXTMANUALDRAG.DRAGOVER
FRMNATIVEDANDD2.IMAGE1.DRAGOVER
FRMNATIVEDANDD2.IMAGE1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.LBLDROPPEDTEXT.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.LBLDROPPEDTEXT.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.LBLDROPPEDTEXT.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.LBLDROPPEDTEXT.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.LBLDROPPEDTEXT.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.LBLDROPPEDTEXT.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.LBLDROPPEDTEXT.DRAGOVER
```

Figure 1. With native drag-and-drop, by default, you see the outline of the control you're dragging.



VFP

VFP

```
FRMNATIVEDANDD2.CNTDRAGDROP1.LBLDROPHERE.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.LBLDROPPEDTEXT.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.LBLDROPPEDTEXT.DRAGDROP
FRMNATIVEDANDD2.TXTMANUALDRAG.DRAG: nAction = 1
FRMNATIVEDANDD2.TXTMANUALDRAG.DRAGOVER
FRMNATIVEDANDD2.TXTMANUALDRAG.DRAGOVER
FRMNATIVEDANDD2.IMAGE1.DRAGOVER
FRMNATIVEDANDD2.IMAGE1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
FRMNATIVEDANDD2.CNTDRAGDROP1.DRAGOVER
CNTDRAGDROP.DRAGDROP
```

Figure 2. The code in the drop target's DragDrop event determines what happens when you drop. Here, the caption of a label inside the container changes.

That's really it, for native drag-and-drop. With those PEMs, you can do quite a bit, but you're limited to working with VFP's objects within VFP.

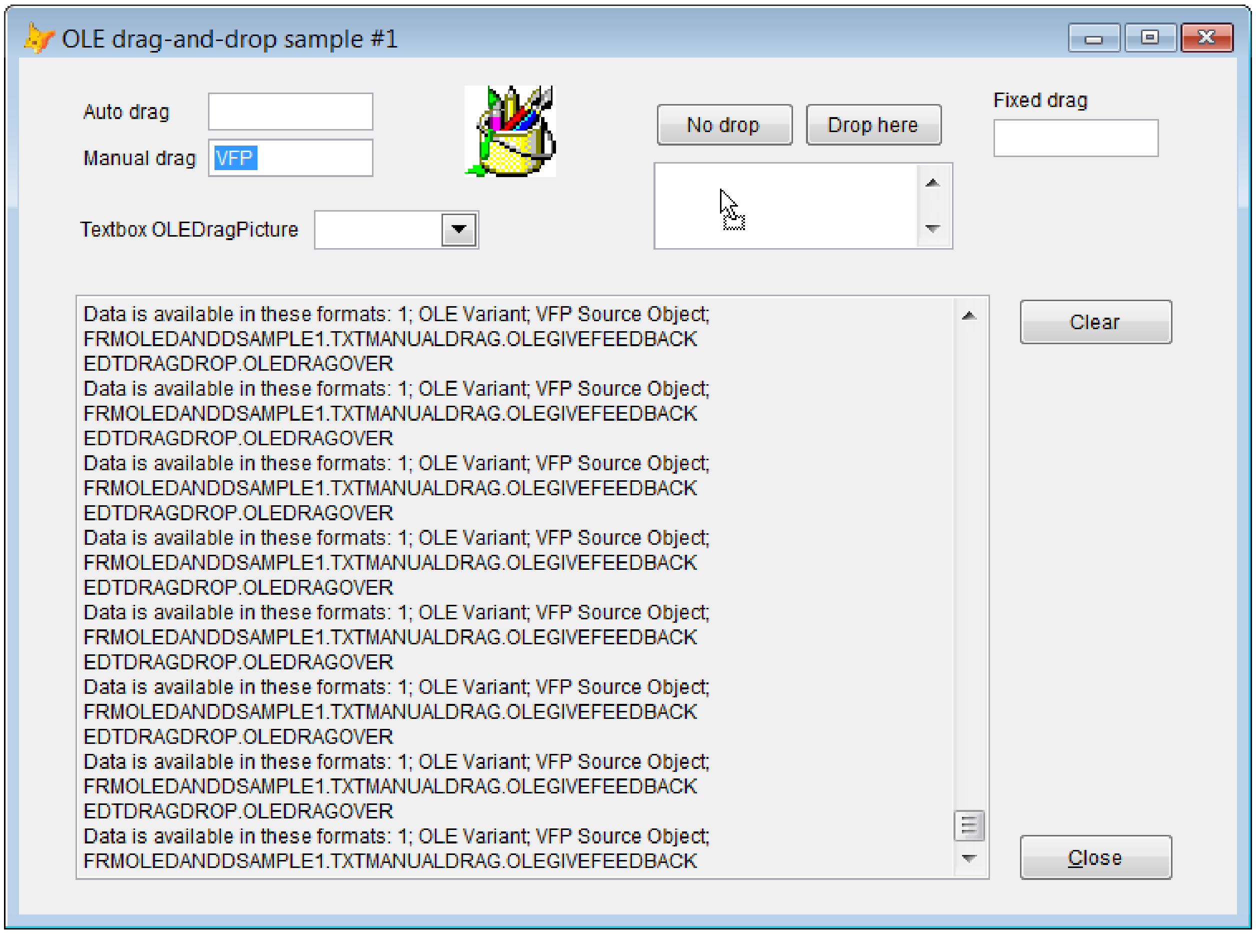
OLE drag-and-drop

OLE drag-and-drop offers a lot more possibilities, both within VFP and when coordinating with other applications. You can drag and drop across applications and you have more control over whichever of the drag source and the drop target are VFP objects.

As with native drag-and-drop, there are two ways to begin OLE dragging a VFP object. You can set OLEDragMode to 1-Automatic. Again like native dragging, dragging begins with pushing the mouse button down and ends when you release the mouse button. However, unlike native drag-and-drop, if you start dragging in a textbox or editbox, nothing happens unless some text is selected. If text is selected, the OLEDrag event fires and dragging begins.

However, there's even less reason to set OLEDragMode to automatic than there is for DragMode. That's because the alternative is to call the OLEDrag method from MouseDown to start dragging. OLEDrag accepts a single logical parameter that indicates whether dragging should start immediately when the method is called or should wait for mouse movement or some time to pass. In other words, it handles automatically what we had to do with code in native drag-and-drop. As a result, it's better to keep OLEDragMode set to 0-Manual, and just put This.OLEDrag(.T.) in the MouseDown event of any control you want to make draggable.

For the simplest cases, that may be all the code you need (except for setting up one or more drop targets by setting their OLEDropMode property). If what you're dragging is text (as from a textbox or editbox) and the target accepts text, the act of dropping is sufficient. The form OLESample1.SCX in the session materials demonstrates. Type something in either of the textboxes, then highlight it and drag it to the smaller editbox. (Be aware that you if you highlight with the mouse, you have to release the mouse button after highlighting and then click it again to being dragging.) When you drop, the text moves from the textbox to the editbox. **Figure 3** shows the process just before dropping.



```
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
```

Figure 3. When you're dragging and dropping text, you may need no code other than a little to start the drag operation. Here, the highlighted text in the "Manual drag" textbox is dragged over the unlabeled editbox.

While the simplicity of dragging text with OLE drag-and-drop is great, what makes it worth learning is everything else you can do. The PEMs involved let you decide what you're actually dragging, what actually gets dropped and how VFP forms and controls behave along the way.

There's really no good order in which to discuss how OLE drag-and-drop works. No matter which PEMs you discuss first, you'll need to refer to stuff that hasn't been discussed yet. Similarly, it's hard to show examples until the whole thing has been discussed. That said, several forms (in particular, OLESample1.SCX and OLESample2.SCX) in this session's downloads demonstrate the various PEMs, and you may want to look at them while reading this description.

OLE drag-and-drop parameters

A number of OLE drag-and-drop methods receive one or both of a pair of parameters. Each of them means the same thing wherever it's used.

The oDataObject parameter is a COM object, described in the section "The Data Object," later in this document, that holds the data being dragged.

The nEffect parameter is an additive value that indicates what can, will or did happen in a drop; the OLEDropEffects property uses the same set of additive values. The list of values is shown in **Table 2**.

Table 2. The nEffect parameter and the OLEDropEffects property get their values by adding one or more of these items together.

Value	Meaning
0	No drop
1	Copy
2	Move
4	Link

Unlike the way most VFP methods work, some of the parameters passed to OLE drag-and-drop methods are passed by reference, so changes within the method have an effect on the operation. Among the things you can change this way are the effect of a drop and the icon used for dragging.

Drag source PEMs

The drag source has one method, four events and a property to control its behavior. The method, OLEDrag, lets you start dragging. The OLEDragPicture property and the OLEGiveFeedback event together determine what the user sees as the object is dragged. In OLE drag-and-drop, the image you see has two parts, one indicating the drag source and one indicating what will happen if you drop on the current drop target. OLEDragPicture determines the drag source part of the image. You can change this property while dragging, but the actual icon displayed doesn't change until the current drag ends. You can set the result portion in the OLEGiveFeedback event. It receives the nEffect parameter so you can tell how the current drop target would deal with a drop; use that information to change the value of the eMouseCursor parameter. You'll rarely need to use OLEGiveFeedback because for the most part, the prospective drop targets provide the necessary feedback.

The OLEStartDrag event fires as dragging begins. To specify what the drop target can do with the data being dragged, set the nEffect parameter in the method. In addition, as described in the subsection, “The Data Object,” later in this section, you can put code in OLEStartDrag to specify what data is being dragged.

The drag source’s OLESetData event gives you one last chance to change the data to be dropped. It fires after the drop target’s OLEDragDrop method, but before the drag source’s OLECompleteDrag. It receives two parameters: the data object and the requested format. See the subsection, “The Data Object,” later in the section, for a discussion of how to modify the contents of the data object.

The drag source’s OLECompleteDrag event is the last to fires in the whole sequence. It receives the nEffect parameter to tell you what actually happened.

Drop target PEMS

The drop target has two events and three properties to control the operation. The OLEDropMode property determines whether the object pays any attention to the drag at all. If it’s set to 0, the object doesn’t respond to the drag and the “no drop” icon shows. Setting OLEDropMode to 1 makes the object a drop target and allows its events to fire. Setting OLEDropMode to 2 says that any drag or drop on this object should instead be handled by the object’s container. See the section “Moving actual objects,” later in this paper, for an example.

The OLEDropHasData property indicates whether the drop target can accept data from the drag source. The default value of -1 lets the objects figure this out on their own, but you can change the property to indicate that the drop target can’t accept data from this drag source (0) or that it can (1). Most of the time, you can leave the default value, but when you want to do something that’s not built in, this property is part of the solution.

The OLEDropEffects property lets the drop target decide how a drop will be handled. As noted earlier, it’s additive, using the values shown in **Table 2**.

OLEDragOver and OLEDragDrop are the OLE drag-and-drop equivalents of the native DragOver and DragDrop events. They fire when you drag over the drop target and when you drop, respectively. They both receive a whole bunch of parameters, as shown in **Listing 4**. The lists are the same, except that OLEDragOver receives an addition parameter, nState, that indicates whether the mouse is entering (0), leaving (1), or remaining within (2) the object.

Listing 4. Both OLEDragOver and OLEDragDrop receive a whole bunch of parameters.

```
PROCEDURE oObject.OLEDragOver
LPARAMETERS oDataObject, nEffect, nButton, nShift, nXCoord, nYCoord, nState

PROCEDURE oObject.OLEDragDrop
LPARAMETERS oDataObject, nEffect, nButton, nShift, nXCoord, nYCoord
```

The oDataObject parameter is discussed in the next subsection, “The Data Object.” The nEffect parameter is the same here as for other OLE drag-and-drop methods.

The nButton and nShift parameters tell you which mouse buttons and which modifier keys were used for the drag. Both are additive, so that multiple buttons and multiple keys can be detected. **Table 3** shows the values for nButton, while **Table 4** lists the values for nShift.

Table 3. The nButton parameter of OLEDragOver and OLEDragDrop tells you which buttons were used for the drag operation. The values of the buttons in use are added together.

Value	Button
1	Left button
2	Right button
4	Middle button

Table 4. The nShift parameter indicates which modifier keys were pressed. The values of the keys pressed are added together.

Value	Key
1	Shift key
2	Ctrl key
4	Alt key

The nXCoord and nYCoord parameters provide the mouse position, relative to the form, at the time the event fired. In some cases, you may want to use them to figure out exactly where to put the dropped item.

As the preceding discussion indicates, a lot happens during OLE drag-and-drop. **Table 5** shows the events that fire in the order they occur. The same drag source is used throughout a single operation, but events of many different drop targets may fire. In addition, the OLEDragOver and OLEGiveFeedback events fire over and over (though possibly for many different objects) as long as dragging continues.

Table 5. The firing order of OLE drag-and-drop methods gives you a lot of ways to control the overall operation.

Method	Parameters	Object	Comments
OLEDrag	lDetectDrag	Drag source	Fires whether using automatic or manual dragging.
OLEStartDrag	oDataObject, nEffect	Drag source	Allows you to prevent the drag or determine the effect of a drop. Allows you to specify the data to be dragged.
OLEDragOver	oDataObject, nEffect, nButton, nShift, nXCoord, nYCoord, nState	Drop target	Fires repeatedly as drag continues. Allows you to inquire about or specify data that would be dropped. Allows you to specify drop target behavior.

Method	Parameters	Object	Comments
OLEGiveFeedback	nEffect, eMouseCursor	Drag source	Fires repeatedly as drag continues. Allows you to change cursor image used for drag.
OLEDragDrop	oDataObject, nEffect, nButton, nShift, nXCoord, nYCoord	Drop target	Fires when mouse button is released. Allows you to inquire about data and respond.
OLESetData	oDataObject, uFormat	Drag source	Allows you to change the data for the specified format.
OLECompleteDrag	nEffect	Drag source	Parameter indicates what actually happened.

At first glance, the list in the table may seem like overkill. Why do we need both OLEDragOver and OLEGiveFeedback? Why both OLEDragDrop and OLESetData? The key is to remember that for some OLE drag-and-drop operations, only one of the drag source and drop targets is a VFP object, and thus only some of these events fire. For example, in the form OLESample2.SCX in this session's downloads, you can drag a filename into Windows Explorer to copy a file. In that case, only the drag source events fire because VFP doesn't control the drop target. (Presumably, Windows Explorer has analogues of these events that fire, but you can't change their code.)

The Data Object

While both the drag source and the drop target have lots of PEMs for OLE drag-and-drop, the key to the whole thing is a third object, called the *data object*. It's passed as a parameter to a number of methods of both the drag source and the drop target and it has methods of its own that let you specify what kind of thing you're dragging and how it should behave.

You can't create the data object or subclass it (it's a COM object), and you don't have access to it except in those methods that receive it as a parameter. But that's sufficient to provide tremendous control over the drag-and-drop process.

The data object contains data from the drag source in one or more formats. **Table 6** lists the available formats. As the table indicates, the format type can be numeric or character, and you can create your own custom formats, using any numeric or character value not already in use. Some other applications may have their own custom formats, which you can use, if you know about them.

Table 6. The data object contains data from the drag source in one or more formats.

Constant (in FoxPro.h)	Value	Description
CF_TEXT	1	Text data
CF_OEMTEXT	7	Text using the OEM character set
CF_UNICODETEXT	13	Text in Unicode format
CF_FILES or CF_HDROP	15	A handle to a list of files
CF_LOCALE	16	A handle to the locale identifier for the text on the clipboard

Constant (in FoxPro.h)	**Value**	**Description**
CFSTR_OLEVARIANTARRAY	“OLE Variant Array”	A VFP array, used for multiple data items.
CFSTR_OLEVARIANT	“OLE Variant”	A VFP variant, which means data in its original format.
CFSTR_VFPSOURCEOBJECT	“VFP Source Object”	A reference to a VFP object
	Any other number or string	Custom data in a format you or another application determines.

When you start dragging, the data object is created and populated with data in one or more formats, depending on what you’re dragging. But you can intercede and change that data or specify there should be data in additional formats. You can also ask the data object whether it has data in a particular format and ask it to give you the data in a particular format. The data object has five methods, shown in **Table 7**. Use GetFormat and GetData to find out what’s being dragged. Use SetFormat and SetData to specify what’s being dragged. (These methods mean that you can make OLE drag-and-drop behave in absurd ways, by replacing what the user thinks she’s dragging with pretty much anything you want.)

Table 7. The data object has five methods you can call to specify what kind of data it makes available and to get your hands on the data.

Method	**Purpose**
ClearData	Remove all data and formats from the data object
GetData	Get data from the data object in a specified format
GetFormat	Determine whether the data object has data in a specified format
SetData	Put data in a specified format into the data object
SetFormat	Indicate that the data object has data in a specified format

You can call the data object’s methods from any of the drag source or drop target methods that receive it as a parameter. For example, you can call SetData from the OLEStartDrag method if you want to provide the same data regardless of the drop target. You can call GetFormat and GetData from the drop target’s OLEDragOver or OLEDragDrop methods to determine what’s available to them and to take action. You can call SetData from the drag source’s OLESetData method to provide data at the last minute, perhaps based on properties set in the OLEDragDrop method.

GetFormat and SetFormat each accept a single parameter, the name or number of the desired format.

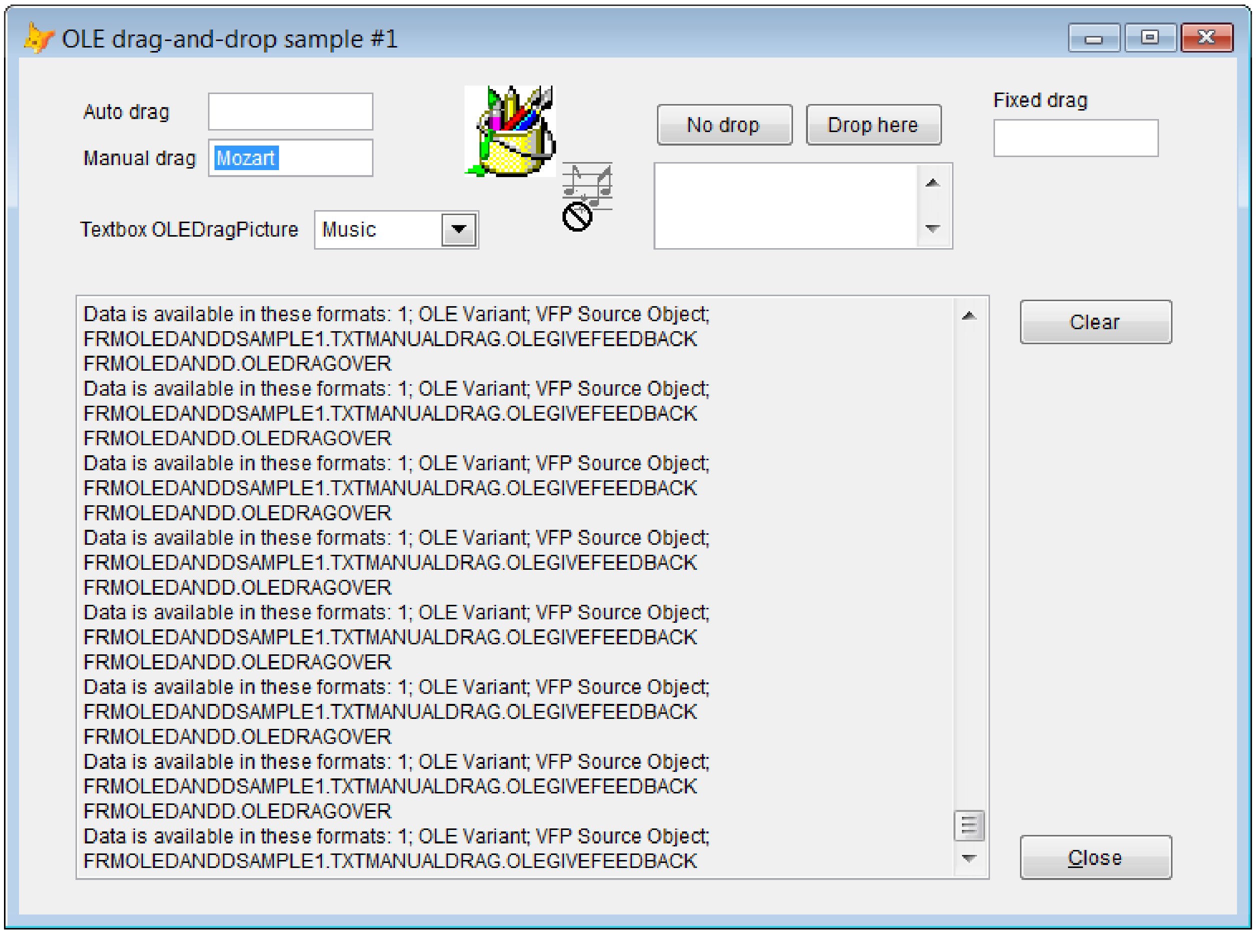
GetData has two parameters. The first is the format of the data you want. The second is an array to hold the data coming back; it’s used only for some data formats, including format 15, where the array is filled with a list of file names. SetData also expects two parameters: the data to store (which also can be an array) and the format.

Putting OLE Drag-and-drop together

Now that we’ve seen all the pieces, we can look at how they fit together through examples. The form shown in **Figure 3** demonstrates several features beyond dragging and dropping text. The dropdown labelled Textbox OLEDragPicture contains the code in **Listing 5** in its Valid method. After making a choice from the dropdown, put some text in either textbox and drag it; **Figure 4** shows what a drag looks like after selecting the music icon; note that you see both the specified drag icon and the appropriate drop icon.

Listing 5. Change OLEDragPicture to control what the user sees while dragging.

```
ThisForm.txtAutoDrag.OLEDragPicture = This.List[This.ListIndex, 2]
ThisForm.txtManualDrag.OLEDragPicture = This.List[This.ListIndex, 2]
```



```
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
```

Figure 4. When you change OLEDragPicture for a drag source, while dragging, you see both the specified icon and the appropriate icon to indicate the effect of a drop.

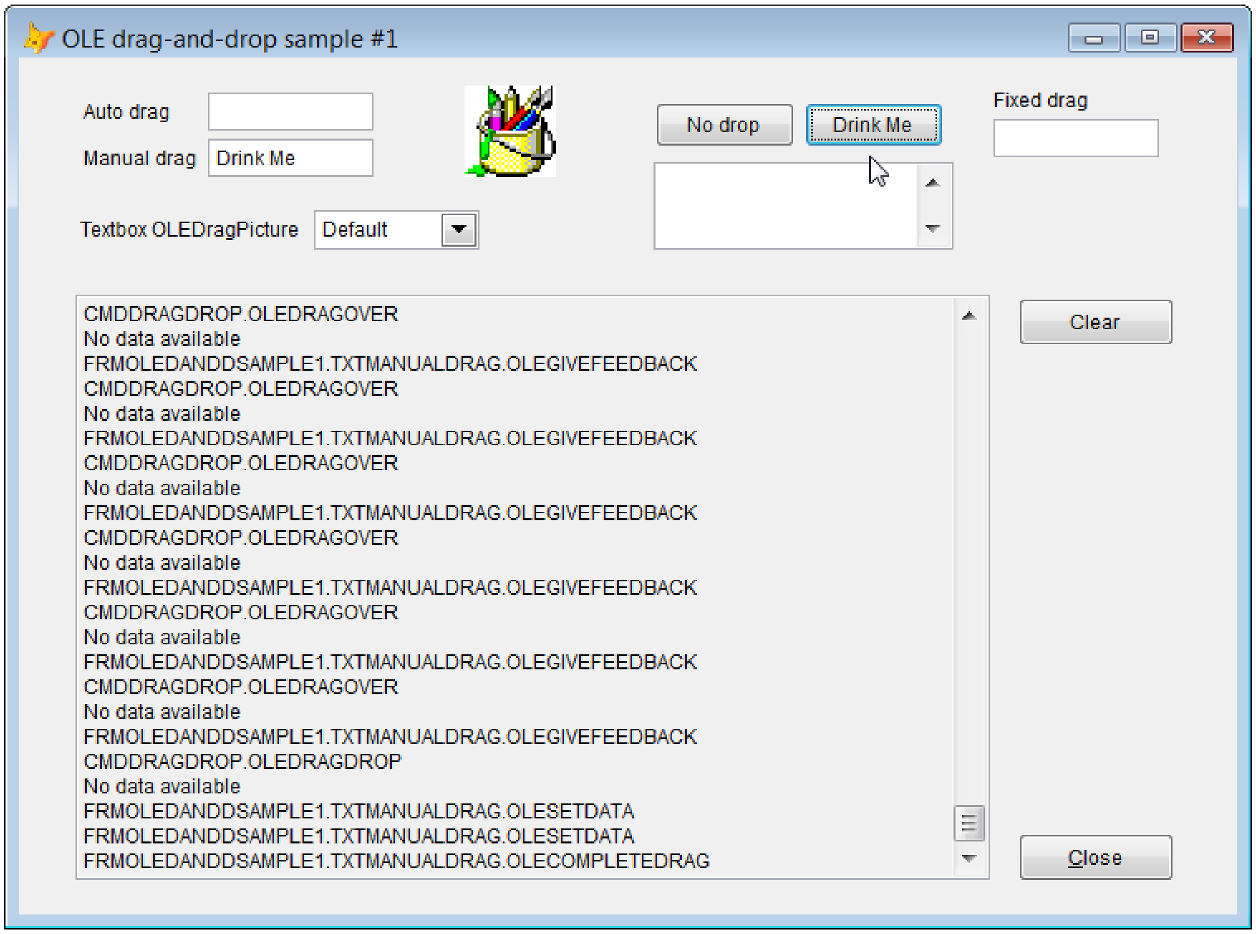
The button with the caption "Drop here" has code in both OLEDragOver and OLEDragDrop; when you drop text on it, the button's caption changes. The OLEDragOver code in **Listing 6** indicates that the button can accept a drop of text (the assignment to OLEDropHasData) and that a drop should result in copying (rather than moving) the text (the assignment to OLEDropEffects). The OLEDragDrop code in **Listing 7** gets the text from the drop and assigns it to the button's caption. **Figure 5** shows the form right after dropping text on the button. At first glance, this may seem like a silly thing to do, but in fact, many modern interfaces let users drag text in this way. For example, Excel's Pivot Table Wizard involves dragging field names to determine what data is used for pivoting.

Listing 6. This code in a button's OLEDragOver method sets things up to allow the button to handle dropped text.

```
IF oDataObject.GetFormat(1)
   This.OLEDropHasData = 1
   This.OLEDropEffects = 1
ENDIF
```

Listing 7. This code in a button's OLEDragDrop method changes the button's caption to the dropped text.

```
IF This.OLEDropHasData = 1
   LOCAL cTextData
   cTextData = oDataObject.GetData(1)
   IF NOT EMPTY(m.cTextData)
      This.Caption = m.cTextData
   ENDIF
ENDIF
```



Manual drag: Drink Me
Textbox OLEDragPicture: Default

```
CMDDRAGDROP.OLEDRAGOVER
No data available
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
CMDDRAGDROP.OLEDRAGOVER
No data available
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
CMDDRAGDROP.OLEDRAGOVER
No data available
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
CMDDRAGDROP.OLEDRAGOVER
No data available
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
CMDDRAGDROP.OLEDRAGOVER
No data available
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
CMDDRAGDROP.OLEDRAGOVER
No data available
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLEGIVEFEEDBACK
CMDDRAGDROP.OLEDRAGDROP
No data available
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLESETDATA
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLESETDATA
FRMOLEDANDDSAMPLE1.TXTMANUALDRAG.OLECOMPLETEDRAG
```

Figure 5. The text in the manual drag textbox has just been dropped on the right-hand button, which changed its caption to match.

It turns out using some controls as drop targets in this way causes problems. See "Form close problems," later in this paper for details.

The image control on the form shows how different drop targets can behave differently with the same drag source. Dropping the image on the form changes the form's icon, while dropping it into the editbox adds the path to the image's picture into the editbox. In addition, while dragging, the image's picture is shown. Code in five methods makes all that work. First, the Image control's OLEStartDrag method (shown in **Listing 8.** contains one line to set OLEDragIcon.

Listing 8. The Image control's OLEDragIcon is set to match its Picture in the control's OLEDragIcon method.

```
This.OLEDragPicture = This.Picture
```

The two possible drop targets each have code in OLEDragOver, shown in **Listing 9**, to indicate that they can accept data from a drag source that offers "VFP Source Object" type data and has a Picture property. This code means that dragging the Image or a button or any other control with a Picture property over the form or the edit box shows that a drop is possible.

Listing 9. The potential drop targets for the Image have code in OLEDragOver that indicates they can accept drops from any drag source that have "VFP Source Object" data, where that VFP source object has a Picture property.

```
LOCAL oSource

IF oDataObject.GetFormat("VFP Source Object")
   oSource = oDataObject.GetData("VFP Source Object")
   IF PEMSTATUS(oSource, 'Picture', 5)
      This.OLEDropHasData = 1
   ENDIF
ENDIF
```

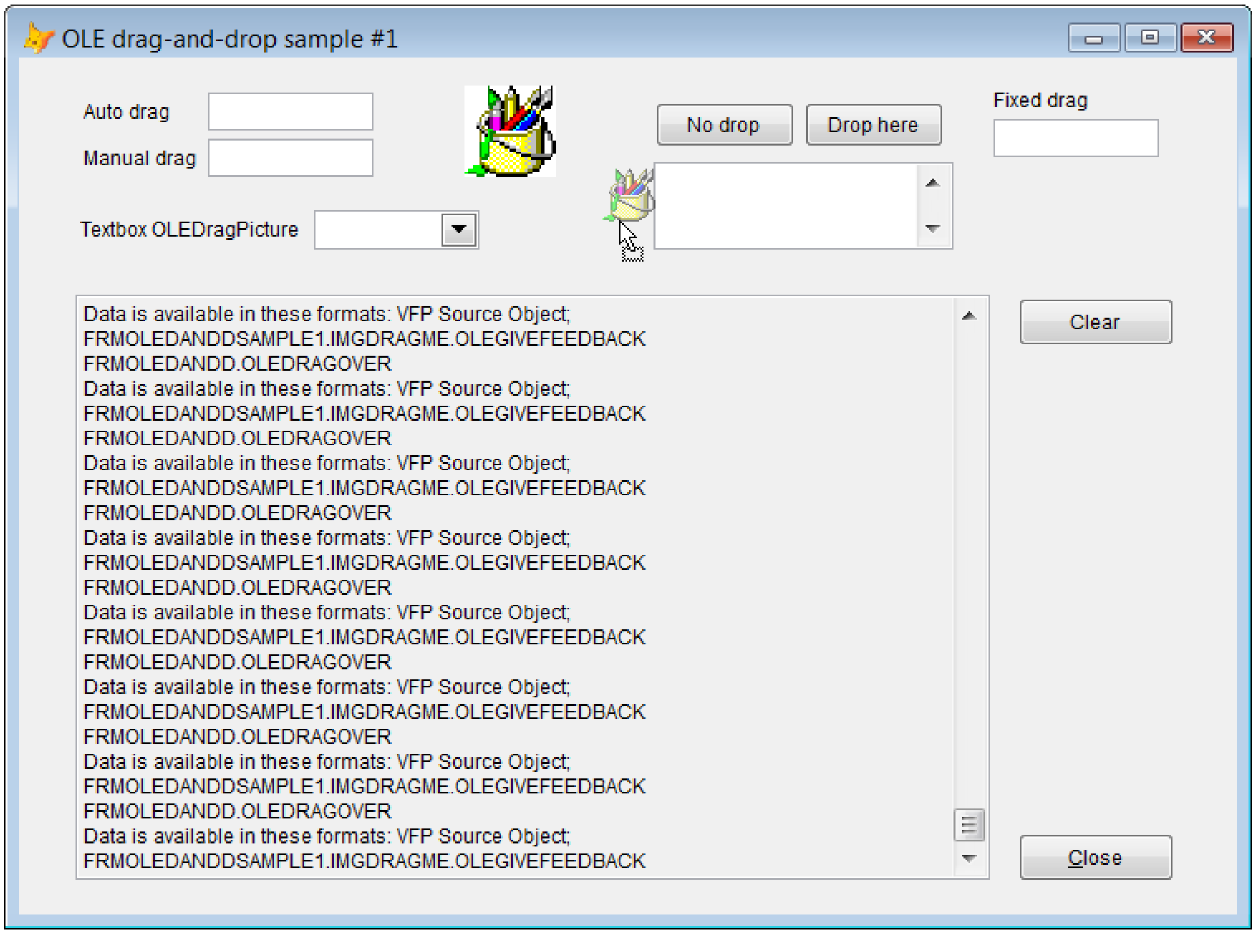
Finally, the two drop targets have code in OLEDragDrop to retrieve the Picture property and do something with it. Because they do different things with it, the code is slightly different for each. **Listing 10** shows the form's OLEDragDrop method, while **Listing 11** shows the code from the editbox's OLEDragDrop. **Figure 6** shows the form as the image is being dragged, while **Figure 7** shows the form after dragging the image twice, dropping it once on the form itself and once in the empty editbox. In practice, you might want to test not just for the presence of the Picture property, but whether or not it's empty. Similarly, you might check the base class of the drag source or its name or anything else, because the "VFP Source Object" data type gives you access to the whole drag source.

Listing 10. This code, in the form's OLEDragDrop method, retrieves the Picture property from the Image (or any other control with a Picture property) and assigns it to the form's Icon property.

```
LOCAL oSource

IF oDataObject.GetFormat("VFP Source Object")
   oSource = oDataObject.GetData("VFP Source Object")
   IF PEMSTATUS(oSource, "Picture", 5)
      This.Icon = oSource.Picture
   ENDIF
ENDIF
```

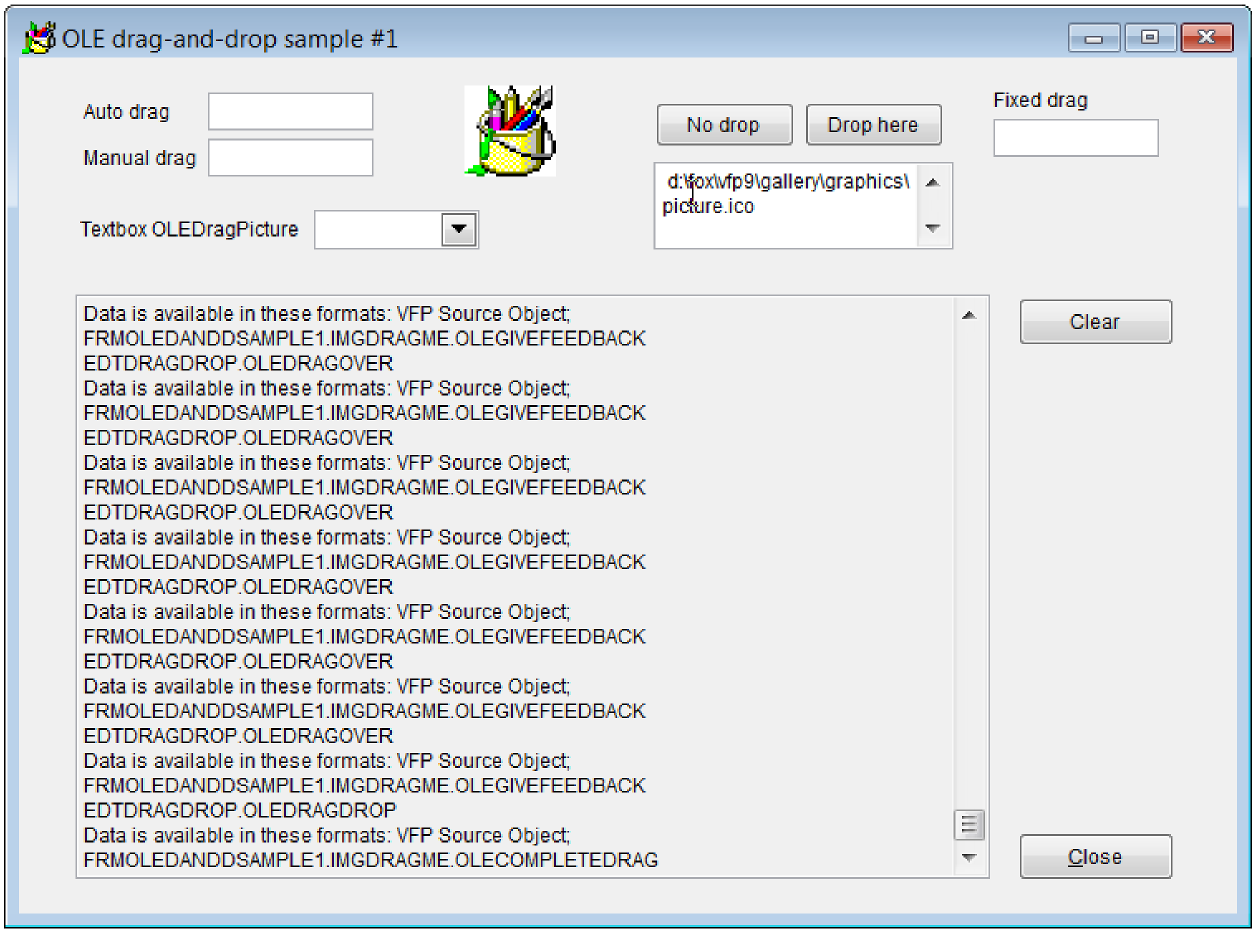
Listing 11. The editbox's OLEDragDrop property contains this code to add the path to the Image's picture to the editbox.

```
LOCAL oSource

IF oDataObject.GetFormat('VFP Source Object')
   oSource = oDataObject.GetData('VFP Source Object')
   IF PEMSTATUS(oSource, 'Picture', 5)
      This.Value = This.Value + ' ' + oSource.Picture
   ENDIF
ENDIF
```



```
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
FRMOLEDANDD.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
```

Figure 6. When dragging the image, it uses its own picture to show what will be dropped.



```
d:\fox\vfp9\gallery\graphics\
picture.ico
```

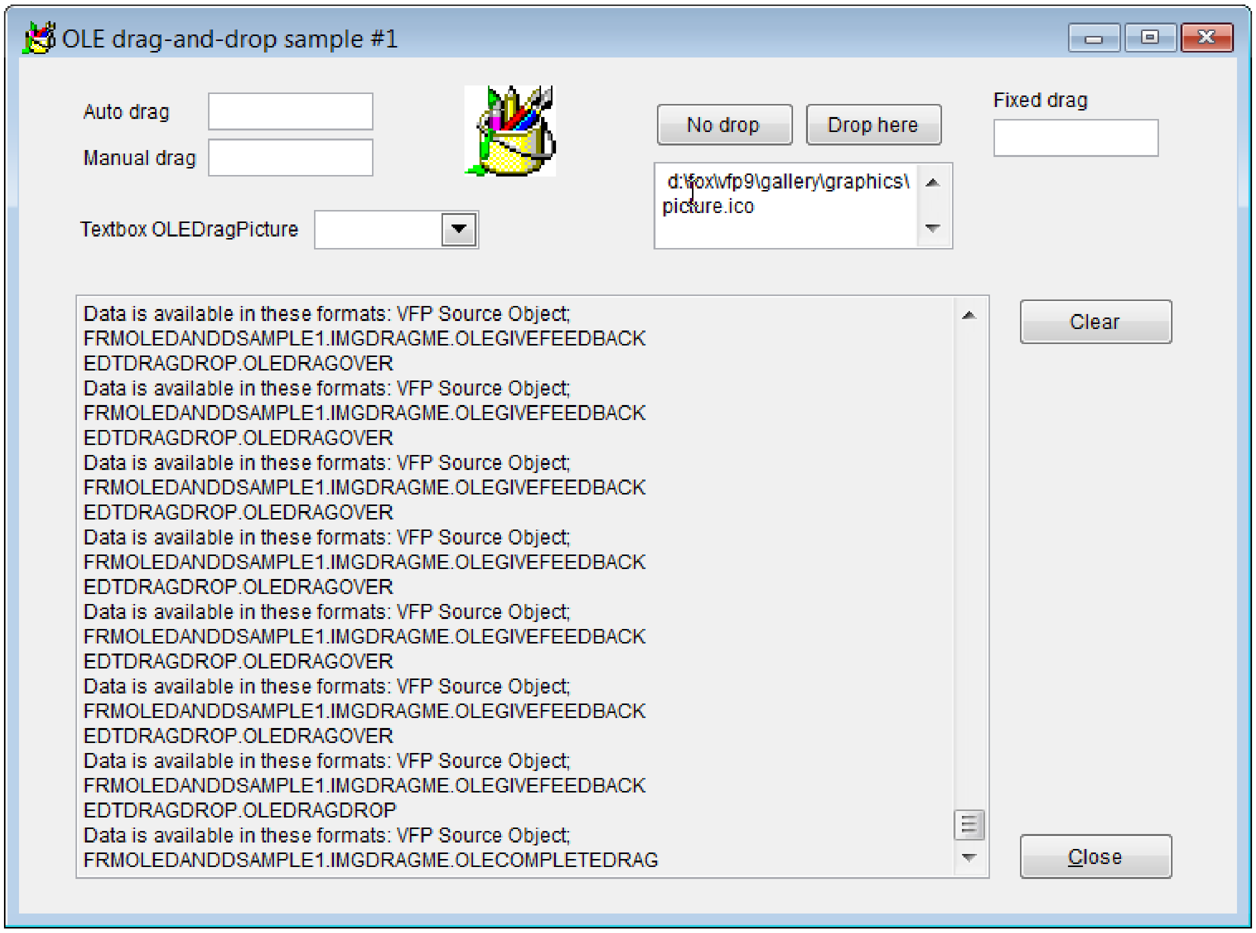
```
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGDROP
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLECOMPLETEDRAG
```

Figure 7. After dragging and dropping the image onto the form and then dragging and dropping it into the editbox, the form's Icon has changed and the path to the image is shown in the editbox.

The final control on the first sample form, the textbox labeled "Fixed drop," has code in its OLESetData method (shown in **Listing 12**) that changes its text data to the string "Haha!" In other words, when you drop text dragged from this control, no matter what you've typed in, the string the drop target receives is "Haha!" **Figure 8** shows the form as the control is being dragged; note there's no sign that the dropped string will be anything other than the text from the control being dragged. **Figure 9** shows the form after the drop.

Listing 12. The OLESetData method of the drag source lets you change the data it provides just before dropping.

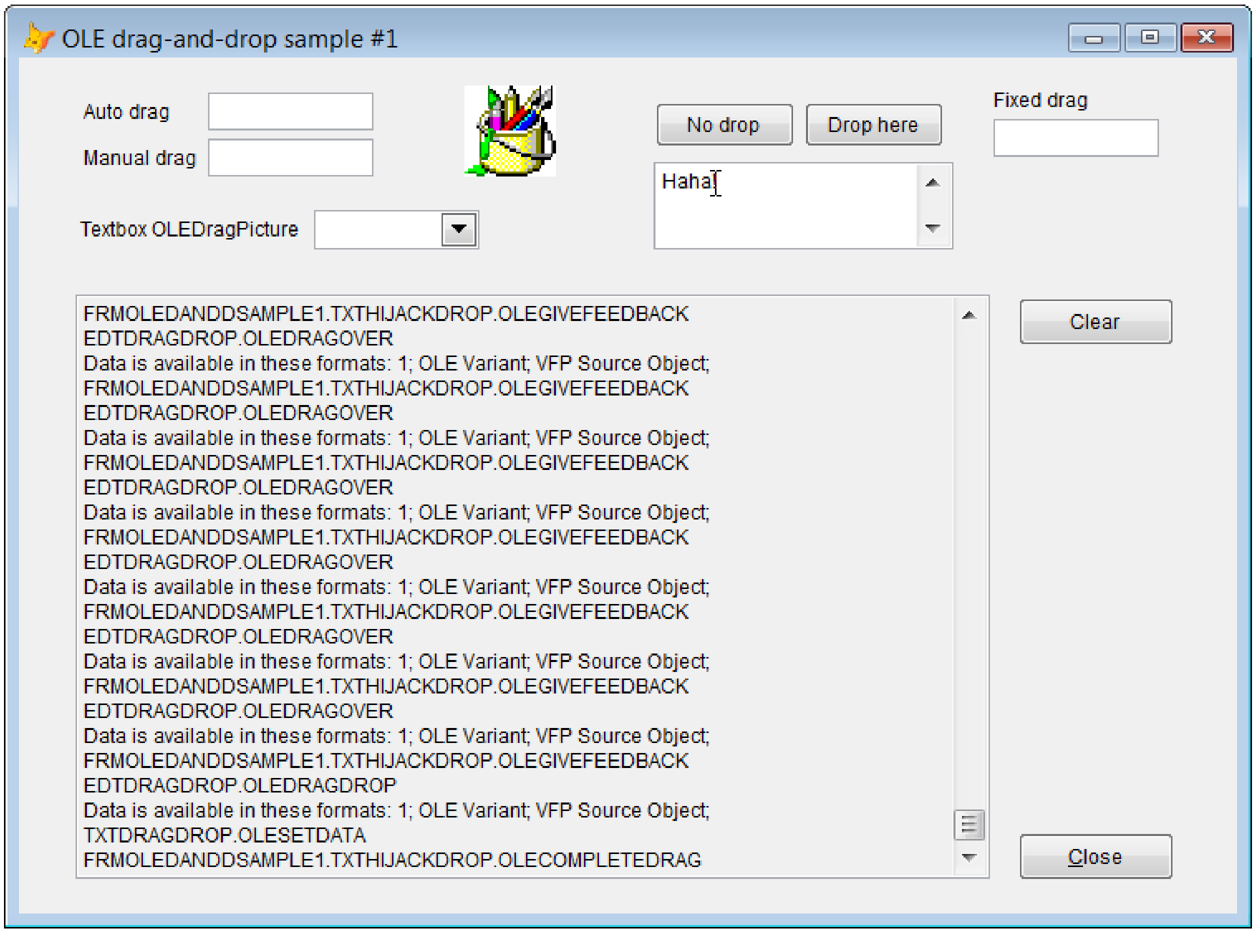
```
oDataObject.SetData("Haha!", 1)
```



```
d:\fox\vfp9\gallery\graphics\picture.ico
```

```
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGDROP
Data is available in these formats: VFP Source Object;
FRMOLEDANDDSAMPLE1.IMGDRAGME.OLECOMPLETEDRAG
```

Figure 8. Just before dropping from the Fixed drag textbox into the editbox, there's no sign that the text will be changed.



```
FRMOLEDANDDSAMPLE1.TXTHIJACKDROP.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTHIJACKDROP.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTHIJACKDROP.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTHIJACKDROP.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTHIJACKDROP.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTHIJACKDROP.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; OLE Variant; VFP Source Object;
FRMOLEDANDDSAMPLE1.TXTHIJACKDROP.OLEGIVEFEEDBACK
EDTDRAGDROP.OLEDRAGDROP
Data is available in these formats: 1; OLE Variant; VFP Source Object;
TXTDRAGDROP.OLESETDATA
FRMOLEDANDDSAMPLE1.TXTHIJACKDROP.OLECOMPLETEDRAG
```

Figure 9. After the drop, the editbox contains "Haha!" rather than "Something"

While making a change like this may seem frivolous or even dangerous, imagine using it to make corrections or add punctuation. You might ensure that the dragged string is surrounded by spaces or quotation marks, for example.

One of the key features of OLE drag-and-drop is the ability to drag between applications. The form in **Figure 10** (OLESample2.SCX in the materials for this session) lets you choose a file and then drag the filename to Windows Explorer; when you do so, the file is copied into the current folder in Explorer. Setting this up requires code in only a single method, OLEStartDrag for the textbox, shown in **Listing 13**.

Listing 13. This code, in the OLEStartDrag method of a textbox, provides file handle data that a drop target can use to access the file.

```
oDataObject.SetFormat(15) && File handles
oDataObject.SetData(ThisForm.cFileName, 15)
```

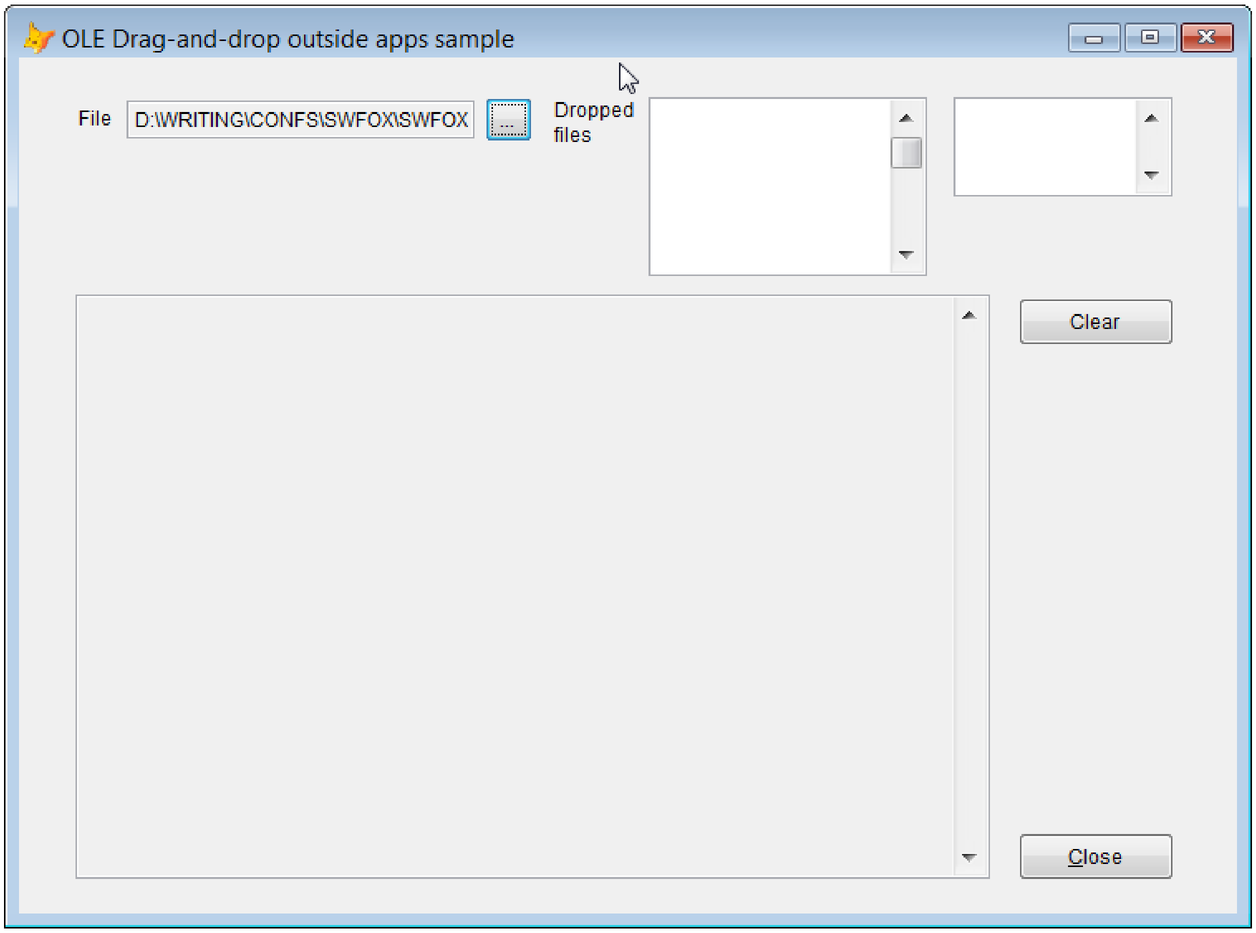

OLE Drag-and-drop outside apps sample
File
D:\WRITING\CONFS\SWFOX\SWFOX
...
Dropped files
Clear
Close

Figure 10. When you drag the filename in the textbox into Windows Explorer, the file is copied to the current folder.

Of course, you can have a VFP drop target for something dragged from outside. The listbox on the same form accepts a drop of files, and puts their names in the list, putting their shared path into the editbox on the side. **Figure 11** shows the form after a drop.

The listbox's OLEDragOver method indicates that it accepts a drop of a list of files, with the code shown in **Listing 14**. OLEDragDrop, in **Listing 15**, contains the code that actually grabs and processes the list of files.

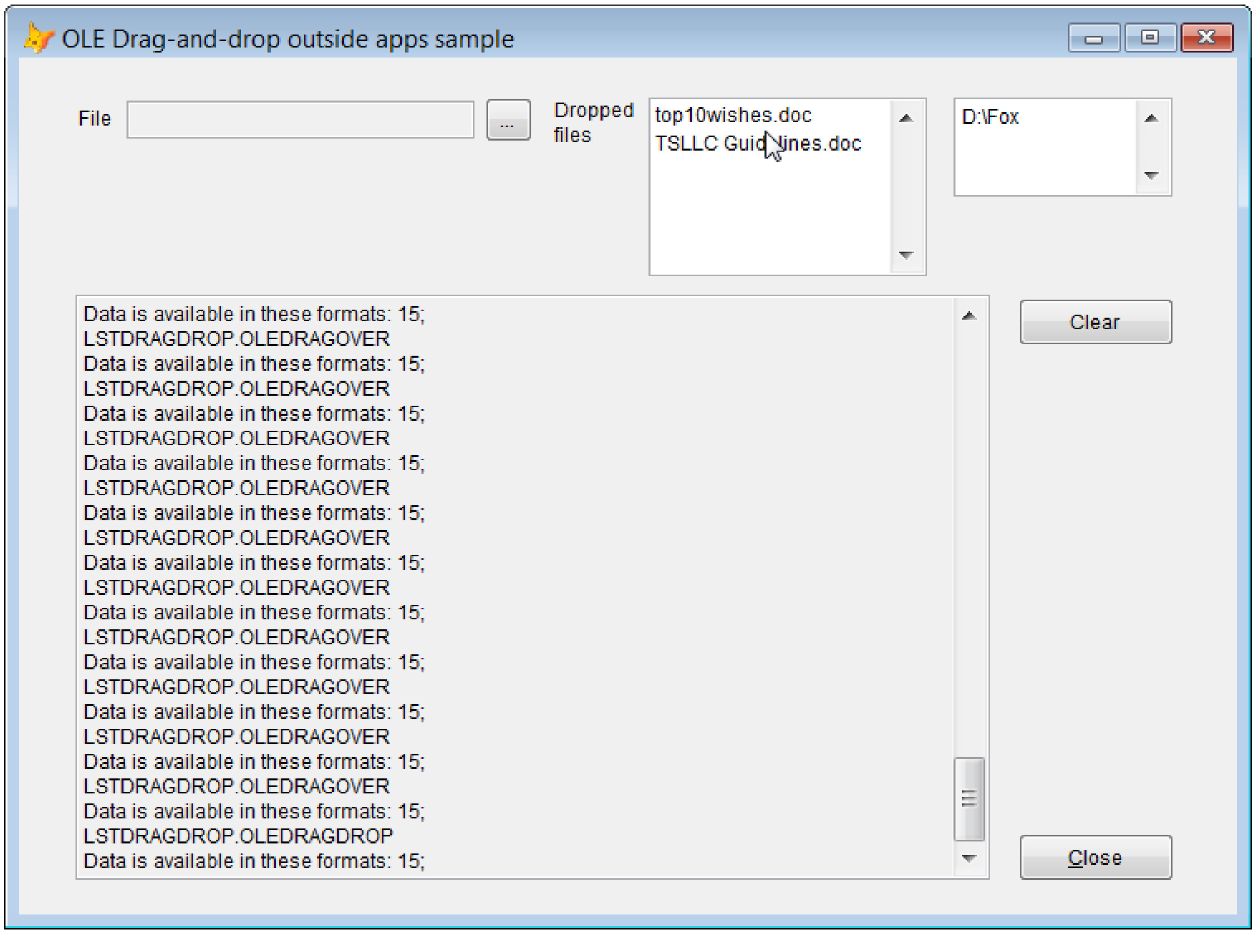
Listing 14. This code in OLEDragOver indicates that the control accepts drops of a list of files, and that the operation used for them is copy.

```
IF oDataObject.GetFormat(15)
   This.OLEDropHasData = 1
   This.OLEDropEffects = 1
ENDIF
```

Listing 15. This code, in OLEDragDrop, gets the list of dragged files and adds them to the list, putting their path into the nearby editbox.

```
LOCAL aFileList[1], nFile, cFile, cPath
```

```
IF oDataObject.GetFormat(15)
   IF oDataObject.GetData(15, @aFileList)
      FOR nFile = 1 TO ALEN(aFileList)
         cFile = aFileList[m.nFile]
         cPath = JUSTPATH(m.cFile)
         This.AddItem(JUSTFNAME(m.cFile))
      ENDFOR

      ThisForm.edtFilePath.Value = m.cPath
   ENDIF
ENDIF
```



top10wishes.doc
TSLLC Guidelines.doc

D:\Fox

```
Data is available in these formats: 15;
LSTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 15;
LSTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 15;
LSTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 15;
LSTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 15;
LSTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 15;
LSTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 15;
LSTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 15;
LSTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 15;
LSTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 15;
LSTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 15;
LSTDRAGDROP.OLEDRAGDROP
Data is available in these formats: 15;
```

Figure 11. The editbox labeled Dropped files accepts a drop from Windows Explorer (or any other application that provides a set of file handles).

Rather than just grabbing the file names, the code in DragDrop could do some kind of processing of each file. That processing might be based on the file type; you have the whole VFP language at your disposal.

Putting Drag-and-drop to work

With the basics covered, let's look at some examples of drag-and-drop in use. The first example here uses native drag-and-drop while the rest use OLE drag-and-drop.

A two-column mover

The two-column mover is a standard interface item, with two listboxes containing items for selection and a variety of ways to move items between the two; one of those ways is by dragging items from one list to the other. I use a mover based on a class created by Marcia Akins and published in "1001 Things You Wanted to Know about Visual FoxPro." In the form shown in **Figure 12** (included in the downloads for this session as moversample.scx), the class is used to select customers.

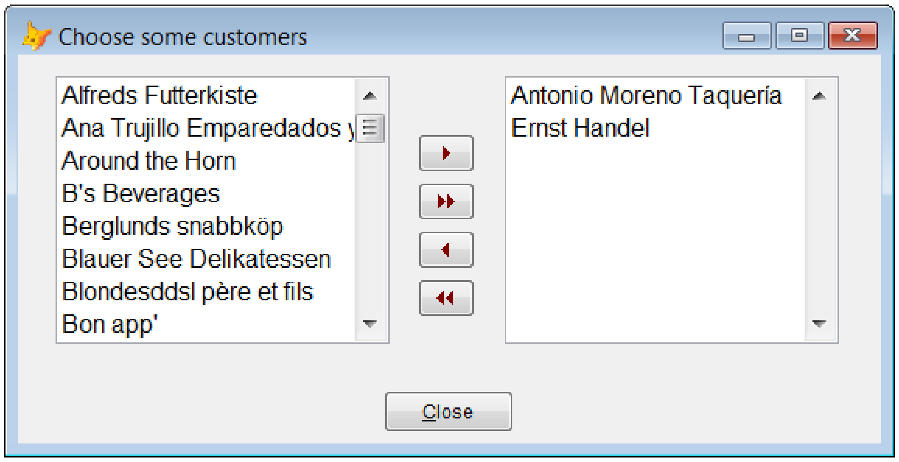


Figure 12. The two-column mover, here used to select customers, lets you drag customers between columns.

All the drag-and-drop code is in the class cntMover, included in the materials for this session in the class library controls.vcx. The container holds two listboxes (lstSource and lstDestination), plus four buttons. Both lists support dragging and dropping to move items between the two lists.

Each list has code in MouseDown, shown in **Listing 16**, to set up the test for a drag starting. The lists' MouseMove methods then delegate the actual test to the container with the code in **Listing 17**.

Listing 16. Each listbox in the mover saves the mouse position at the container level to determine whether a drag is underway.

```
THIS.Parent.nMouseX = nXCoord
THIS.Parent.nMouseY = nYCoord
```

Listing 17. This code in the listboxes's MouseMove method lets the mover decide whether has begun.

```
IF nButton = 1
   This.Parent.StartDrag( This, nXCoord, nYCoord )
ENDIF
```

Each list's DragOver method calls the container's custom ChangeIcon method, as in **Listing 18.** That method, shown in **Listing 19**, checks whether the mouse is entering or leaving the list and sets the icon accordingly.

Listing 18. The DragOver methods of the two listboxes call on the container to set the drag icon.

```
This.Parent.ChangeIcon( oSource, nState )
```

Listing 19. The custom ChangeIcon method of the container is called from the two listboxes' DragOver method. It changes the drag icon based on whether the mouse is entering or leaving a listbox.

```
LPARAMETERS toSource, tnState
IF tnState = 0
   *** allowed to drop
   toSource.DragIcon = THIS.cDropIcon
ELSE
   IF tnState = 1
      *** not allowed to drop
      toSource.DragIcon = THIS.cNoDropIcon
   ENDIF
ENDIF
```

Finally, the DragDrop method of the two listboxes (shown in **Listing 20**) calls the container's MoveItems method to do the work of moving items from one list to the other. MoveItems is the same method called by the Move and Remove buttons and by the lists' DblClick methods.

Listing 20. The DragDrop method of each listbox confirms that it's not the drag source and then calls the mover's custom MoveItems method to do the actual work.

```
IF oSource.Name # THIS.Name
   THIS.Parent.MoveItems( oSource )
ENDIF
```

Moving actual objects

In an application I wrote for a client some years ago, a form (see **Figure 13**) shows circuit boards (the gray boxes with hexagons at the top) in a multiplexer. The application's specs included the ability to drag the circuit boards from one slot to another. (Because this is client code, I can't include the form in the conference materials.)

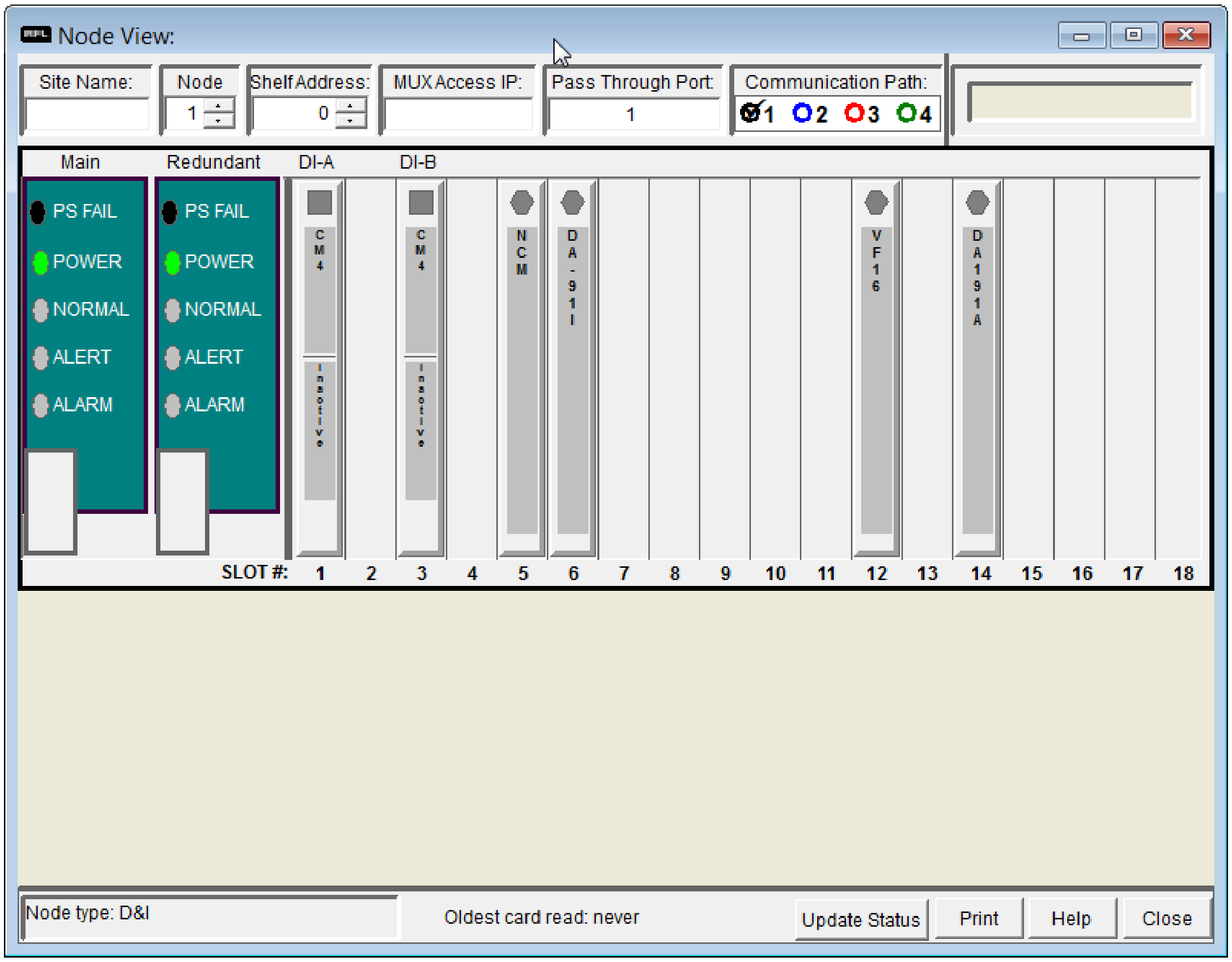


Figure 13. The gray vertical objects represent circuit boards in a multiplexer. The application specs called for the ability to drag them from one slot to another.

It takes very little code to make it work. The circuit board is represented by a class called cntCard. The MouseDown method of cntCard calls a custom method named StartCardDrag. StartCardDrag, in turn, calls OLEDrag(.T.) to start dragging.

The container for the cards is a class named cntSlots. Among the things it contains is a shape named shpSlotSpace. That shape's OLEDropMode is set to 2-Pass to container, so that cntSlots can handle drops. Note that OLEDropMode is to 0-Disabled for cntCard; that prevents dropping one circuit board on another.

The OLEDragOver method of cntSlots, shown in Error! Not a valid bookmark self-reference., makes the drop possible by setting the OLEDropHasData property.

Listing 21. The OLEDragOver property of the cntSlots object (the one that contains all the circuit boards) indicates that a drop can be accepted.

```
IF oDataObject.GetFormat("VFP Source Object")
   This.OLEDropHasData = 1
ENDIF
```

Finally, the OLEDragDrop method of cntSlots calls a custom form method, MoveCard, passing the drag source, the container for cntSlots (an object of a class called cntShelf) and the position of the drop; the code is shown in **Listing 22**. The form's MoveCard method is complex, because it not only moves the physical object (cntCard), but figures out what the underlying business objects are and calls on them to make the necessary changes to the object hierarchy.

Listing 22. This code, in cntSlots' OLEDragDrop method, asks the form to do the actual work of moving the circuit board from one slot to another.

```
IF oDataObject.GetFormat("VFP Source Object")
   ThisForm.MoveCard(oDataObject.GetData("VFP Source Object"), ;
                     This.Parent, nYCoord, nXCoord)
ENDIF
```

Dragging from Outlook

It's easy to imagine an application where users want to drag items from Outlook and have them processed in some way. For example, you might want users to drag a contact from Outlook and have the application add that contact to a table, send an email to that contact, or perhaps make that contact the designated contact for a particular project. You might want to drag an appointment from Outlook to set the date for some application item.

Unfortunately, by default, dropping an Outlook item onto a drop target (that accepts text) simply copies a few fields and their column headers from the item. While other Office applications receive Outlook items in something resembling their native format, VFP can't access that format, even if you set it up as a custom format.

Instead, in order to actually access the Outlook data, you need to combine OLE drag-and-drop with Automation. (My thanks to Ben Creighton and Derek Jackson, who provided a roadmap to this approach in their contributions to the topic "Outlook Drag Drop" in the Visual FoxPro Wiki: http://fox.wikis.com/wc.dll?Wiki~OutlookDragDrop~WIN_COM_API.) This solution isn't perfect (in particular, you can't distinguish between dragging an email item and dragging an attachment to that item), but for most purposes, it's good enough.

The key is that to drag from Outlook, it must be running, and thus you can not only connect via Automation, but in a drag-and-drop situation, you can pay attention to what's currently selected, which must be what was dragged. To do so, you use Outlook's ActiveExplorer property, which provides a reference to the Outlook interface.

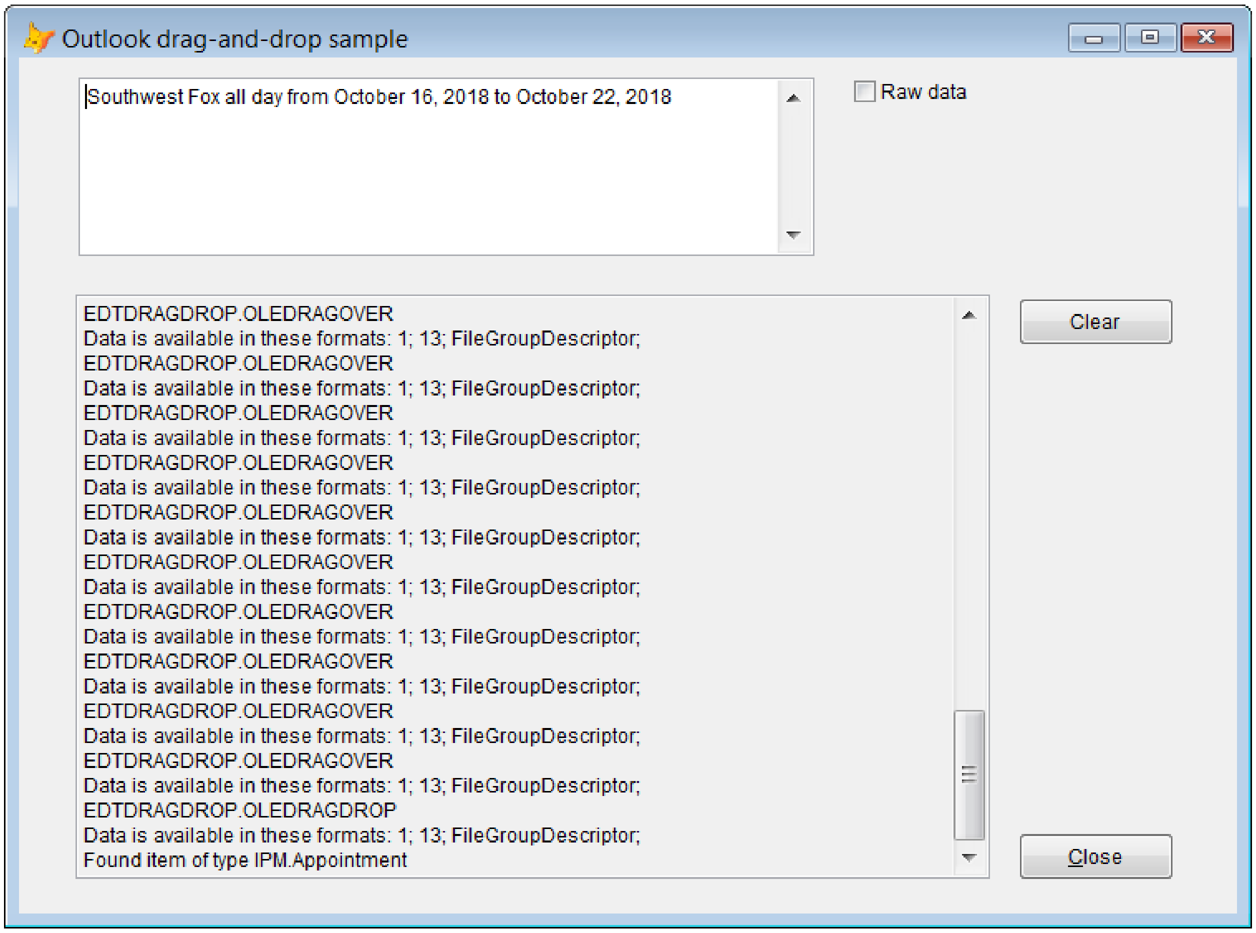
You need a way to hold onto various references to Outlook. I chose to create a custom class with a reference to Outlook itself and to several of its key objects. The class is included in the materials for this session as OutlookApp.PRG; the key code is in the Init method, shown in **Listing 23**.

Listing 23. This code in the Init method of the custom OutlookApp class, grabs references to several key Outlook objects.

```
PROCEDURE Init
```

```
LOCAL lSuccess

TRY
   This.oOutlook = CREATEOBJECT('Outlook.Application')
   This.oNameSpace = This.oOutlook.GetNameSpace('MAPI')
   This.oFolders = This.oNameSpace.Folders
   This.oExplorer = This.oOutlook.ActiveExplorer
   lSuccess = .T.
CATCH
   MESSAGEBOX("Unable to connect to Outlook. Cannot drag into this form from 
Outlook")
   lSuccess = .F.
ENDTRY

RETURN m.lSuccess
```

Any form that accepts drops from Outlook needs a reference to this object. The form shown in **Figure 14** has a custom oOutlook property. (It's included in the downloads for this session as OLESampleOutlook.SCX.) Code in the form's Init method instantiates the OutlookApp class and stores a reference in the oOutlook property. (This means that ThisForm.oOutlook.oOutlook is the form's reference to Outlook itself.)



```
Southwest Fox all day from October 16, 2018 to October 22, 2018
```

```
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; 13; FileGroupDescriptor;
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; 13; FileGroupDescriptor;
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; 13; FileGroupDescriptor;
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; 13; FileGroupDescriptor;
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; 13; FileGroupDescriptor;
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; 13; FileGroupDescriptor;
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; 13; FileGroupDescriptor;
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; 13; FileGroupDescriptor;
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; 13; FileGroupDescriptor;
EDTDRAGDROP.OLEDRAGOVER
Data is available in these formats: 1; 13; FileGroupDescriptor;
EDTDRAGDROP.OLEDRAGDROP
Data is available in these formats: 1; 13; FileGroupDescriptor;
Found item of type IPM.Appointment
```

Figure 14. This form combines Automation with OLE drag-and-drop to capture dropped Outlook data. Here, the multi-day all-day event for Southwest Fox has been dropped into the editbox.

As always, to do anything more than dropping text, it takes code in both OLEDragOver and OLEDragDrop. The editbox's OLEDragOver method, shown in **Listing 24**, checks for data in a useful format, including the format Outlook uses, FileGroupDescriptor.

Listing 24. This code, in OLEDragOver, checks for data in a format of interest, including Outlook's FileGroupDescriptor format.

```
IF oDataObject.GetFormat("FileGroupDescriptor")
   This.OLEDropHasData = 1
   This.OLEDropEffects = 1 && copy
ENDIF
```

If the OLEDragDrop method finds FileGroupDescriptor data, it asks Outlook about the currently selected item and processes it, as in **Listing 25**. The checkbox on the form turns off this behavior and allows the default behavior for a drop from Outlook.

First, we confirm we have FileGroupDesciptor data. If we do and we have a reference to Outlook, we get a reference to the currently selected item or items (ThisForm.oOutlook.oExplorer.Selection). Outlook hands back a collection of selected items, so we loop through the collection. The MessageClass property of an item indicates what type of item it is, as the CASE statement shows. The code here simply displays some information about the item in the editbox; in an application, you'd likely do more with that data.

Listing 25. The editbox's OLEDragDrop method asks Outlook for its current item and processes that in response to the drop.

```
LOCAL oObjColl, oObj, cInfo, nAttachment, cCentury

IF NOT Thisform.chkRawData.Value
   IF oDataObject.GetFormat("FileGroupDescriptor")
      IF NOT ISNULL(ThisForm.oOutlook)
         oObjColl = ThisForm.oOutlook.oExplorer.Selection
         FOR EACH oObj IN oObjColl
            ThisForm.Addmessage("Found item of type " + oObj.MessageClass)
            DO CASE
            CASE oObj.MessageClass = "IPM.Note"
               * Email or attachment
               * Can't tell one from the other
               cInfo = oObj.Subject + CHR(13) + CHR(10)
               cInfo = m.cInfo + " From: " + oObj.Sender.Name + CHR(13) + CHR(10)
               cInfo = m.cInfo + " To: " + oObj.To + CHR(13) + CHR(10)
               cInfo = m.cInfo + " Received: " + TRANSFORM(oObj.ReceivedTime) + ;
                       CHR(13) + CHR(10)

               IF oObj.Attachments.Count > 0
                  cInfo = cInfo + " Atttachments: " + ;
                          TRANSFORM(oObj.Attachments.Count) + CHR(13) + CHR(10)
                  nAttachment = 0
                  FOR EACH oAtt IN oObj.Attachments
                     nAttachment = m.nAttachment + 1
                     cInfo = m.cInfo + "  " + TRANSFORM(m.nAttachment) + ": " + ;
                             oAtt.FileName + CHR(13) + CHR(10)
                  ENDFOR
               ENDIF

            CASE oObj.MessageClass = "IPM.Appointment"
               * Calendar item
               cInfo = oObj.Subject
               nLength = oObj.End - oObj.Start
               cCentury = SET("Century")
               SET CENTURY ON
               IF oObj.AllDayEvent
                  cInfo = m.cInfo + " all day "
                  IF m.nLength = 24 * 60 * 60
                     cInfo = m.cInfo + "on " + MDY(TTOD(oObj.Start))
                  ELSE
                     cInfo = m.cInfo + "from " + MDY(TTOD(oObj.Start)) + " to " + ;
                             MDY(TTOD(oObj.End-1))
                  ENDIF
               ELSE
                  cInfo = m.cInfo + " from " + TTOC(oObj.Start) + " to " + ;
                          TTOC(oObj.End)
               ENDIF
               SET CENTURY &cCentury

            CASE oObj.MessageClass = "IPM.Contact"
               * Contact item
               cInfo = oObj.FullName + CHR(13) + CHR(10)
               IF NOT EMPTY(oObj.HomeAddress)
                  cInfo = m.cInfo + " Home: " + oObj.HomeAddress + CHR(13) + CHR(10)
               ENDIF
               IF NOT EMPTY(oObj.BusinessAddress)
                  cInfo = m.cInfo + " Work: " + oObj.BusinessAddress + ;
                          CHR(13) + CHR(10)
               ENDIF
               IF NOT EMPTY(oObj.HomeTelephoneNumber)
                  cInfo = m.cInfo + " Home: " + oObj.HomeTelephoneNumber + ;
                          CHR(13) + CHR(10)
               ENDIF
               IF NOT EMPTY(oObj.BusinessTelephoneNumber)
                  cInfo = m.cInfo + " Work: " + oObj.BusinessTelephoneNumber + ;
                          CHR(13) + CHR(10)
               ENDIF
               IF NOT EMPTY(oObj.CarTelephoneNumber)
                  cInfo = m.cInfo + " Mobile: " + oObj.CarTelephoneNumber + ;
                          CHR(13) + CHR(10)
               ENDIF
               IF NOT EMPTY(oObj.Email1Address)
                  cInfo = m.cInfo + " Email: " + oObj.Email1Address + ;
                          CHR(13) + CHR(10)
               ENDIF

            CASE oObj.MessageClass = "IPM.Task"
               * Task item
               cInfo = oObj.Subject  + CHR(13) + CHR(10)
               IF NOT EMPTY(oObj.DueDate)
                  cInfo = cInfo + " Due: " + TTOC(oObj.DueDate)  + CHR(13) + CHR(10)
               ENDIF
               cInfo = m.cInfo + IIF(oObj.Complete, ' ', ' NOT ') + "complete"

            OTHERWISE
               * Unknown
               cInfo = "Sorry. I couldn't understand what you dropped."

            ENDCASE

            This.Value = This.Value + m.cInfo  + CHR(13) + CHR(10)

         ENDFOR
      ENDIF
   ENDIF
ENDIF
```

There are a few weaknesses here. First, as noted, you can't distinguish an email item from an attachment to that item. (The code here reports on both.) In addition, there's no guarantee that Outlook is the only application providing data in the FileGroupDescriptor format. I haven't found any other, but that doesn't mean they don't exist.

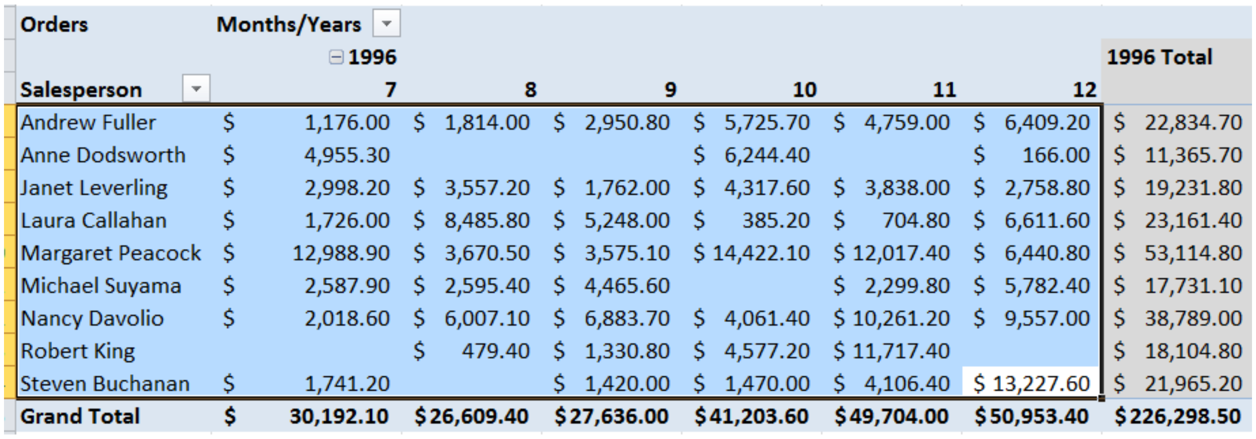
Dragging between grids and Office

While dragging from Outlook into VFP is tricky, dragging between Excel or Word and VFP is simple. It takes only a little code to drag an Excel selection or the contents of a Word table into a VFP grid, or to put VFP grid contents into Excel or Word.

Dragging from Excel or Word into a grid

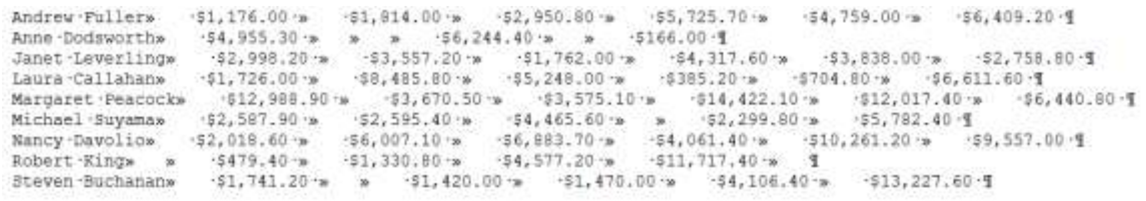
Excel and Word make their contents available in a number of formats, but the easiest to work with is format 1-text. (For this example, we're considering a selected table in Word. In general, you can drag and drop text from Word directly into VFP's text controls without any code.)

In both cases, the tabular contents are packaged with tabs between columns and returns between lines. For example, **Figure 15** shows part of a pivot table of Northwind sales by employee by month. **Figure 16** shows what you get when you drag the highlighted portion into a text file. (Note that dragging highlighted data out of Excel is a little tricky. You have to move the mouse over the edge of the selected area until you see the resize icon and then click and drag.)



Orders	Months/Years						
	1996						1996 Total
Salesperson	7	8	9	10	11	12	
Andrew Fuller	$ 1,176.00	$ 1,814.00	$ 2,950.80	$ 5,725.70	$ 4,759.00	$ 6,409.20	$ 22,834.70
Anne Dodsworth	$ 4,955.30			$ 6,244.40		$ 166.00	$ 11,365.70
Janet Leverling	$ 2,998.20	$ 3,557.20	$ 1,762.00	$ 4,317.60	$ 3,838.00	$ 2,758.80	$ 19,231.80
Laura Callahan	$ 1,726.00	$ 8,485.80	$ 5,248.00	$ 385.20	$ 704.80	$ 6,611.60	$ 23,161.40
Margaret Peacock	$ 12,988.90	$ 3,670.50	$ 3,575.10	$ 14,422.10	$ 12,017.40	$ 6,440.80	$ 53,114.80
Michael Suyama	$ 2,587.90	$ 2,595.40	$ 4,465.60		$ 2,299.80	$ 5,782.40	$ 17,731.10
Nancy Davolio	$ 2,018.60	$ 6,007.10	$ 6,883.70	$ 4,061.40	$ 10,261.20	$ 9,557.00	$ 38,789.00
Robert King		$ 479.40	$ 1,330.80	$ 4,577.20	$ 11,717.40		$ 18,104.80
Steven Buchanan	$ 1,741.20		$ 1,420.00	$ 1,470.00	$ 4,106.40	$ 13,227.60	$ 21,965.20
Grand Total	$ 30,192.10	$26,609.40	$27,636.00	$41,203.60	$49,704.00	$50,953.40	$226,298.50

Figure 15. This spreadsheet holds a pivot table of Northwind sales by employee by month. The names and the 1996 data are selected.



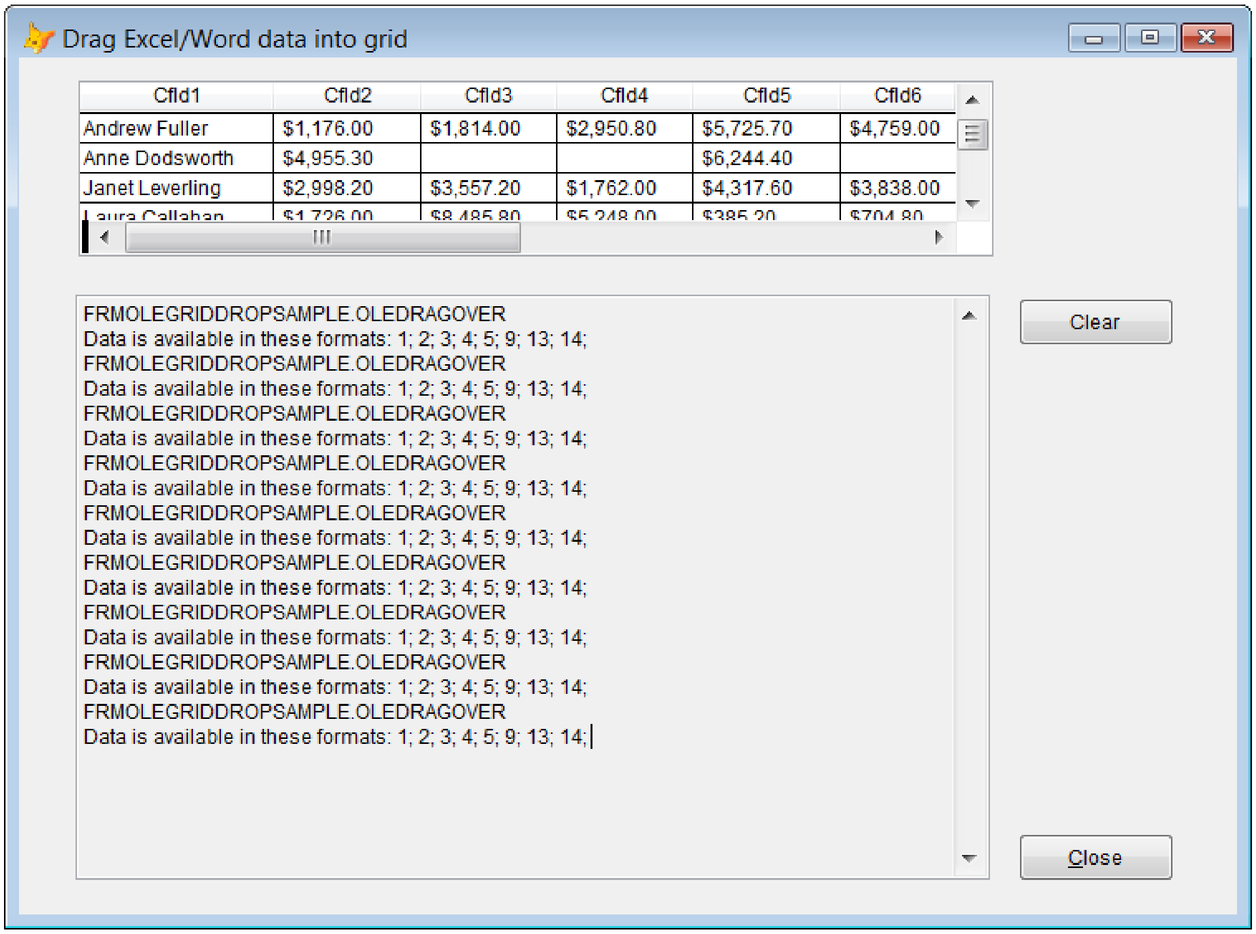
```
Andrew·Fuller»    ·$1,176.00·»    ·$1,814.00·»    ·$2,950.80·»    ·$5,725.70·»    ·$4,759.00·»    ·$6,409.20·¶
Anne·Dodsworth»   ·$4,955.30·»  »   »   ·$6,244.40·»   »   ·$166.00·¶
Janet·Leverling»   ·$2,998.20·»    ·$3,557.20·»    ·$1,762.00·»    ·$4,317.60·»    ·$3,838.00·»    ·$2,758.80·¶
Laura·Callahan»   ·$1,726.00·»    ·$8,485.80·»    ·$5,248.00·»    ·$385.20·»    ·$704.80·»    ·$6,611.60·¶
Margaret·Peacock»    ·$12,988.90·»    ·$3,670.50·»    ·$3,575.10·»    ·$14,422.10·»    ·$12,017.40·»    ·$6,440.80·¶
Michael·Suyama»   ·$2,587.90·»    ·$2,595.40·»    ·$4,465.60·»   »   ·$2,299.80·»    ·$5,782.40·¶
Nancy·Davolio»   ·$2,018.60·»    ·$6,007.10·»    ·$6,883.70·»    ·$4,061.40·»    ·$10,261.20·»    ·$9,557.00·¶
Robert·King»   »   ·$479.40·»    ·$1,330.80·»    ·$4,577.20·»    ·$11,717.40·»   ¶
Steven·Buchanan»   ·$1,741.20·»   »   ·$1,420.00·»    ·$1,470.00·»    ·$4,106.40·»    ·$13,227.60·¶
```

Figure 16. Data dragged out of Excel is tab-separated with returns at the end of each line.

Parsing data in that format isn't hard. The most challenging part is figuring out how many columns there are and how wide each needs to be. **Figure 17** shows a form (OLESampleGrid2.SCX in the session materials) with a grid that does the necessary parsing. The grid's OLEDragOver method, shown in **Listing 26**, is similar to previous examples. It checks for text data and, if it finds it, indicates the grid can accept the drop as a copy.

Listing 26. The grid's OLEDragOver method checks for text data.

```
IF oDataObject.GetFormat(1)
   This.OLEDropHasData = 1
   This.OLEDropEffects = 1
ENDIF
```



Cfld1	Cfld2	Cfld3	Cfld4	Cfld5	Cfld6
Andrew Fuller	$1,176.00	$1,814.00	$2,950.80	$5,725.70	$4,759.00
Anne Dodsworth	$4,955.30			$6,244.40	
Janet Leverling	$2,998.20	$3,557.20	$1,762.00	$4,317.60	$3,838.00
Laura Callahan	$1,726.00	$8,485.80	$5,248.00	$385.20	$704.80

```
FRMOLEGRIDDROPSAMPLE.OLEDRAGOVER
Data is available in these formats: 1; 2; 3; 4; 5; 9; 13; 14;
FRMOLEGRIDDROPSAMPLE.OLEDRAGOVER
Data is available in these formats: 1; 2; 3; 4; 5; 9; 13; 14;
FRMOLEGRIDDROPSAMPLE.OLEDRAGOVER
Data is available in these formats: 1; 2; 3; 4; 5; 9; 13; 14;
FRMOLEGRIDDROPSAMPLE.OLEDRAGOVER
Data is available in these formats: 1; 2; 3; 4; 5; 9; 13; 14;
FRMOLEGRIDDROPSAMPLE.OLEDRAGOVER
Data is available in these formats: 1; 2; 3; 4; 5; 9; 13; 14;
FRMOLEGRIDDROPSAMPLE.OLEDRAGOVER
Data is available in these formats: 1; 2; 3; 4; 5; 9; 13; 14;
FRMOLEGRIDDROPSAMPLE.OLEDRAGOVER
Data is available in these formats: 1; 2; 3; 4; 5; 9; 13; 14;
FRMOLEGRIDDROPSAMPLE.OLEDRAGOVER
Data is available in these formats: 1; 2; 3; 4; 5; 9; 13; 14;
FRMOLEGRIDDROPSAMPLE.OLEDRAGOVER
Data is available in these formats: 1; 2; 3; 4; 5; 9; 13; 14;
```

Figure 17. The grid in this form parses text data and builds a cursor from it, then displays that cursor.

The real work happens in the grid's OLEDragDrop method, shown in **Listing 27**. If there is text data available, GetData is called to get it. Then, the data is put into an array with one row for each row from the original. Each row is parsed to determine how many columns there are. Since different rows in the dragged data may have data in different numbers of columns, each row has to be checked. At the same time, the actual data is checked to determine the longest value in each column.

Once we have all that information, we can create a cursor with the right number of columns and with each column wide enough to hold the largest value. (In this example, I'm creating them all as character. You could write additional code to determine whether all data in any column is of some other type and then use that data type for the column.) Finally, we copy the data from the array to the cursor and set the cursor as the grid's RecordSource. (Note that the code will fail if any data item includes all three of VFP string delimiters: single quotes, double quotes and square brackets.)

Listing 27. The grid's OLEDragDrop method parses the dropped data to find out how many rows and columns there are, and how wide each column needs to be. Then it creates a cursor containing the data.

```
LOCAL aRowData[1], aOneRow[1], aColSize[1], aDraggedData[1]
LOCAL nRows, nCols, nColCount
LOCAL nColWidth
LOCAL nRow, nCol
LOCAL cData, cEndDelimiter, cItem, cStartDelimiter, uData

IF oDataObject.GetFormat(1)
   uData = oDataObject.GetData(1)
   * Parse result into rows and cols and put into a cursor
   nRows = ALINES(aRowData, m.uData)

   * First pass is to determine number of cols
   nColCount = 0
   FOR nRow = 1 TO m.nRows
      nCols = ALINES(aOneRow, aRowData[m.nRow], CHR(9))
      nColCount = MAX(m.nColCount, m.nCols)
      DIMENSION aColSize[m.nColCount]
      FOR nCol = 1 TO m.nCols
         IF VARTYPE(aColSize[m.nCol]) <> 'N'
            aColSize[m.nCol] = 1
         ENDIF
         aColSize[m.nCol] = MAX(aColSize[m.nCol], LEN(aOneRow[m.nCol]))
      ENDFOR
   ENDFOR

   * Now create a cursor
   LOCAL cCursorDef, cFld

   cCursorDef = "CREATE CURSOR csrGridSrc ( "
   FOR nCol = 1 TO m.nColCount
      IF aColSize[m.nCol] > 254
         cFld = "mFld" + TRANSFORM(m.nCol) + " M "
      ELSE
         cFld = "cFld" + TRANSFORM(m.nCol) + ;
                " C(" + TRANSFORM(aColSize[m.nCol]) + ")"
      ENDIF

      cCursorDef = m.cCursorDef + m.cFld
      IF m.nCol < m.nColCount
         cCursorDef = m.cCursorDef + ", "
      ELSE
         cCursorDef = m.cCursorDef + ")"
      ENDIF
   ENDFOR

   &cCursorDef

   * Now populate
   FOR nRow = 1 TO m.nRows
      nCols = ALINES(aOneRow, aRowData[m.nRow], CHR(9))
      cData = ''
      FOR nCol = 1 TO m.nCols
         cItem = aOneRow[m.nCol]
         IF '[' $ m.cItem OR ']' $ m.cItem
            IF ['] $ m.cItem
               cStartDelimiter = ["]
            ELSE
               cStartDelimiter = [']
            ENDIF
         ELSE
            cStartDelimiter = '['
         ENDIF

         IF m.cStartDelimiter = '['
            cEndDelimiter = ']'
         ELSE
            cEndDelimiter = m.cStartDelimiter
         ENDIF

         cData = m.cData + m.cStartDelimiter + aOneRow[m.nCol] +
                 m.cEndDelimiter + ","
      ENDFOR

      * Add any missing columns at the end
      FOR nCol = m.nCol TO m.nColCount
         cData = cData + "[],"
      ENDFOR

      cData = LEFT(m.cData, LEN(m.cData) - 1)

      INSERT INTO csrGridSrc VALUES (&cData)
   ENDFOR

   GO TOP IN csrGridSrc
   This.RecordSource = "csrGridSrc"
ENDIF
```

The same code works for data dragged from a Word table.

Dragging from a grid to Excel or Word

Knowing what format Excel expects makes the reverse problem, dragging from a grid into Excel, easier (with one complication, described a little later). We just have to provide data in the desired format and Excel will do the rest.

The session materials contain OLESampleGridDrag, which displays a grid that you can drag to Excel or Word to copy the data. The grid's OLEStartDrag method, shown in **Listing 28**, packages up the data from the grid's RecordSource into a format that both Excel and Word can handle. The routine loops through the RecordSource and creates a string with columns separated by tabs, and rows separated by CRLF. (Although Excel provides data with CR only, it works fine with CRLF, and Word works better when both are provided.) Although the code here is specific to the data used in the example form, it wouldn't be much harder to write generic code, using the FIELD() function.

Listing 28. This code, in a grid's OLEStartDrag method, packages the grid data so it can be dropped.

```
* Set up data
LOCAL cGridData, nRecNo

cGridData = ''
SELECT (This.RecordSource)
nRecNo = RECNO(This.RecordSource)

SCAN
   cGridData = m.cGridData + ALLTRIM(FirstName) + CHR(9) + ;
               ALLTRIM(LastName) + CHR(9) + ;
               ALLTRIM(Title) + CHR(9) + ;
               TRANSFORM(BirthDate) + CHR(13) + CHR(10)
ENDSCAN

GO (m.nRecNo) IN (this.RecordSource)

oDataObject.SetData(m.cGridData, 1)

RETURN
```

When you drag this data into Word, you don't get a table, but you get a block of text that can be converted to a table using Insert | Table | Convert Text to Table from the Word menu.

The one complication is configuring the grid so you can actually drag it. When you click into a grid, the control onto which you clicked receives the MouseDown event, not the grid itself. In order to have the grid process MouseDown and start the drag, the Init method of the grid class (grdDragDrop in DragDropBase in the session materials) contains the code in **Listing 29**. lDragGrid is a custom property that must be set to .T. to enable dragging of the grid as a whole. The BindMouseDown method, shown in **Listing 30**, drills down through the grid to bind the MouseDown method of every component to the grid itself.

Listing 29. This code in the grid's Init ensures that MouseDown is processed by the grid, not its contained controls

```
IF This.lDragGrid
   This.BindMouseDown(This)
ENDIF
```

Listing 30. The grid's custom BindMouseDown method uses BindEvent to ensure that clicking the mouse anywhere in the grid fires the grid's MouseDown method.

```
LPARAMETERS oContainer

LOCAL oObject

FOR EACH oObject IN oContainer.Objects FOXOBJECT
```

```
   IF PEMSTATUS(m.oObject, "MouseDown", 5)
      BINDEVENT(m.oObject, "MouseDown", This, "MouseDown")
   ENDIF

   IF PEMSTATUS(oObject, "Objects", 5)
      This.BindMouseDown(m.oObject)
   ENDIF
ENDFOR
```

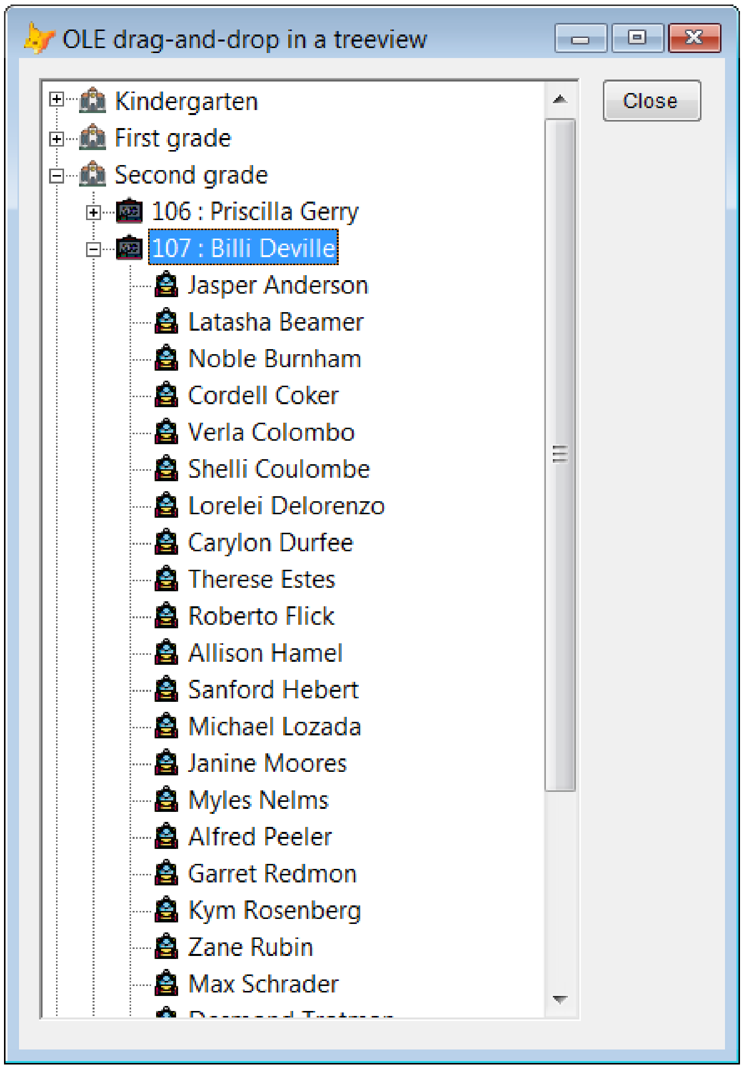
Drag-and-drop in a Treeview

Soon after this session was conceived, Doug Hennig asked whether I was planning to talk about dragging and dropping with TreeView controls. He noted that it's common to want to drag a child node from its parent to another or to drag a file from Explorer to add it.

I'd never tried drag-and-drop with TreeView, but it turned out that not only had Doug done so, but he had worked out the kinks and created a treeview class that makes it all easy. Rather than recreating the wheel, I got his permission to use his class. (Not surprisingly, Doug has written about all this, too. You'll find Doug's papers about treeviews at http://doughennig.com/papers/Pub/200407dhen.pdf, http://doughennig.com/papers/Pub/200408dhen.pdf, and http://doughennig.com/papers/Pub/ExplorerInterfaces.pdf. The classes used for this example come from the third paper; those classes are also included in the downloads for this session.)

The example here uses the sftreeviewcursor class found in sftreeview.vcx. That class is a subclass of sftreeviewcontainer, which contains the basic drag-and-drop code described here.

Figure 18 shows the classes and students in a school organized in a treeview; the form is included in the downloads for this session as TreeviewDragDrop.SCX. The top-level nodes are the grades, the next level contains classrooms (showing the room number and the teacher's name), and the bottom level shows students. The tables that provide the data are included in the materials for this session. (Note that the Students and Teachers table are drawn from sample data originally created for another session, so they contain a lot more data than needed for the example and follow a different set of naming conventions.)



Kindergarten
First grade
Second grade
106 : Priscilla Gerry
107 : Billi Deville
Jasper Anderson
Latasha Beamer
Noble Burnham
Cordell Coker
Verla Colombo
Shelli Coulombe
Lorelei Delorenzo
Carylon Durfee
Therese Estes
Roberto Flick
Allison Hamel
Sanford Hebert
Michael Lozada
Janine Moores
Myles Nelms
Alfred Peeler
Garret Redmon
Kym Rosenberg
Zane Rubin
Max Schrader

Figure 18. This form lets you move a student from one class to another using OLE Drag-and-drop.

Doug's treeview class is actually a container that holds a TreeView, an ImageList, and a couple of other controls. The TreeView, of course, is contained in a OLEControl, and the OLEControl has the usual drag-and-drop methods. Doug's class delegates from those methods to methods of the treeview container. For example, the OLEControl's OLEStartDrag method contains the code in **Listing 31**. MouseDown, OLECompleteDrag, OLEDragDrop and OLEDragOver contain analogous code. All the interesting code is in the container's TreeOLEXXX methods.

Listing 31. The methods of the OLEControl for the treeview, such as OLEStartDrag, delegate to the container that holds it.

```
* Pass OLEStartDrag events to the parent.

lparameters toData, ;
   tnAllowedEffects
This.Parent.TreeOLEStartDrag(@toData, @tnAllowedEffects)
```

Doug's code addresses a variety of issues with treeviews. For example, TreeMouseDown makes sure that the node under the mouse is the one selected before deciding whether to start dragging. In addition, code in TreeDragOver ensures that if you drag near the top or bottom of the tree, it scrolls so you can see items that aren't currently visible.

Three custom methods let you decide whether to allow dragging (CanStartDrag), whether to allow a drop (CanDrop), and what to do with a drop (HandleDragDrop). They're called at the appropriate times and mean that you can, for the most part, just use Doug's code as is. Be aware that CanDrop is called from both TreeOLEDragOver and TreeOLEDragDrop (which are called from the treeview's OLEDragOver and OLEDragDrop methods, respectively).

In the example, there's code in those three methods, as well as the container's FillTreeViewCursor and LoadImages methods. The code in LoadImages, as the name suggests, provides the images used in the treeview; it's shown in **Listing 32**. (The images are included in the downloads for this session; they were all created by Freepik and downloaded from www.flaticon.com.)

Listing 32. This code in the treeview container's LoadImages method sets up the images used for the three levels in the treeview.

```
WITH This.oIMAGELIST
   * Icons made by Freepik from www.flaticon.com
   .ListImages.Add(1, "Grade", LOADPICTURE("School.bmp"))
   .ListImages.Add(2, "Class", LOADPICTURE("Blackboard.bmp"))
   .ListImages.Add(3, "Student", LOADPICTURE("Backpack.bmp"))
ENDWITH
```

Not surprisingly, the code in FillTreeViewCursor gathers the data to display and puts it into a cursor in the format Doug's sftreeviewcursor class expects; it's shown in **Listing 33**. (See Doug's paper for an explanation of the expected format.) The three GetXXX methods called here each run a single query to collect data for that level of the tree.

Listing 33. The custom FillTreeViewCursor method lets you fill a cursor with data the class uses to populate and manage the treeview.

```
* Populate with grades, classes and students
ThisForm.GetGrades()
SCAN
   INSERT INTO (This.cCursorAlias) ;
      (ID, ;
       Type, ;
       Text, ;
       Image, ;
       Sorted) ;
      VALUES ;
      (TRANSFORM(csrGradeList.iGradeID), ;
       'Grade', ;
       csrGradeList.mDesc, ;
       "Grade", ;
       .T.)
```

```
ENDSCAN

ThisForm.GetClasses()
SCAN
   INSERT INTO (This.cCursorAlias) ;
      (ID, ;
       Type, ;
       ParentID, ;
       ParentType, ;
       Text, ;
       Image, ;
       Sorted) ;
      VALUES ;
      (TRANSFORM(csrClasses.iClassID), ;
       'Class', ;
       TRANSFORM(csrClasses.iGradeID), ;
       'Grade', ;
       csrClasses.cRoom + ": " + csrClasses.cName, ;
       'Class', ;
       .f.)
ENDSCAN

ThisForm.GetStudents()
SCAN
   INSERT INTO (This.cCursorAlias) ;
      (ID, ;
       Type, ;
       ParentID, ;
       ParentType, ;
       Text, ;
       Image) ;
      VALUES ;
      (TRANSFORM(csrStudents.StudentID), ;
       'Student', ;
       TRANSFORM(csrStudents.iClassID), ;
       'Class', ;
       csrStudents.cName, ;
       'Student')
ENDSCAN
```

In the example, we want to allow only students to be dragged. So the CanStartDrag method (shown in **Listing 34**) makes sure the node being dragged is a student node. Because sftreeviewcursor maintains the cCurrentNodeType property, it takes only one line of code to figure that out.

Listing 34. The CanStartDrag method is used to determine whether a particular node can be dragged. Here, it limits dragging to Student nodes.

```
* Only students can be dragged
RETURN This.cCurrentNodeType = 'Student'
```

If we're dragging only students, then the only place we can drop is classes; it wouldn't make sense to drop a student onto a grade level. In addition, there's no reason to drop a student onto the class they're currently in. So the code in CanDrop checks those conditions. If they're met, it sets the drop effect to copy + move. In testing, I found that setting it to move only worked badly in cases where drops are allowed by the CanDrop code, but are prevented by additional code in HandleDragDrop.

In addition to the usual OLEDragOver parameters, CanDrop receives an object reference to the selected node (toNode) and to a specially constructed object (toObject) that contains information about the drag source and the drop target. The code in CanDrop is shown in **Listing 35**. The format CF_MAX (17) is one of the Windows' clipboard formats; the TreeOLEStartDrag method in sftreeviewcontainer puts information about the drag source into the data object in that format, as well as text format.

Listing 35. The custom CanDrop method is called by both TreeOLEDragOver and TreeOLEDragDrop to determine whether to allow the drop.

```
lparameters toData,  toNode,  toObject,  tnEffect,  tnButton,  tnShift

* Allow drops only onto classes

LOCAL cDragKey, lResult
LOCAL nRecNo

lResult = .F.

DO CASE
CASE toData.GetFormat(CF_MAX) AND toObject.DragType = 'Student' AND ;
     toObject.DropType = 'Class'
   * Make sure it's a different class than the student
   * is now in
   cDragKey = toObject.DragKey
   nRecNo = RECNO("ClassLists")
   IF SEEK(ALLTRIM(m.cDragKey), "ClassLists", "iStudentID")
      IF VAL(toObject.DropKey) <> ClassLists.iClassID
         lResult = .T.
      ENDIF
   ENDIF
   GO (m.nRecNo) IN ClassLists

OTHERWISE
   lResult = .F.
ENDCASE


IF m.lResult
   tnEffect = DROPEFFECT_MOVE + DROPEFFECT_COPY
ELSE
   tnEffect = DROPEFFECT_NONE
ENDIF

RETURN m.lResult
```

The HandleDragDrop method (shown in **Listing 36**) makes one additional check, whether you're moving the child to a different grade. If so, it prompts to confirm you want to do that. If it's appropriate, it changes the child's classroom assignment and refreshes the tree.

Listing 36. The custom HandleDragDrop method gets the code to actually respond to the drop.

```
lparameters toData,  toNode,  toObject

* If dropped student on a different class, move it.
* Do so by changing the underlying data and then
* reloading
LOCAL lProceed

DO CASE
CASE toData.GetFormat(CF_MAX) AND toObject.DragType = 'Student' AND ;
     toObject.DropType = 'Class'

   lProceed = .T.

   cDragKey = toObject.DragKey
   cDropKey = toObject.DropKey

   * Check student's current grade and the grade of
   * the proposed new class.
   SELECT Grades.iGradeID, mDesc ;
      FROM Grades ;
        JOIN ClassRooms ;
          ON Grades.iGradeID = ClassRooms.iGradeID ;
        JOIN ClassLists ;
          ON Classrooms.iClassID = ClassLists.iClassID ;
      WHERE ClassLists.iStudentID = VAL(m.cDragKey) ;
      INTO CURSOR csrCurGrade

   SELECT Grades.iGradeID, mDesc ;
      FROM Grades ;
        JOIN ClassRooms ;
          ON Grades.iGradeID = ClassRooms.iGradeID ;
      WHERE ClassRooms.iClassID = VAL(m.cDropKey) ;
      INTO CURSOR csrNewGrade

   IF csrCurGrade.iGradeID <> csrNewGrade.iGradeID
      cMessage = "Student is currently in " + ALLTRIM(csrCurGrade.mDesc) + ". " + ;
                 "Do you really want to move the student to " + ;
                 ALLTRIM(csrNewGrade.mDesc) + "?"
      IF MESSAGEBOX( m.cMessage, 4+32, "Move student to new grade") = 6
         lProceed = .T.
      ELSE
         lProceed = .F.
      ENDIF
   ENDIF

   IF m.lProceed
      nRecNo = RECNO("ClassLists")
      IF SEEK(ALLTRIM(m.cDragKey), "ClassLists", "iStudentID")
```

```
         REPLACE iClassID WITH VAL(toObject.DropKey) IN ClassLists
      ENDIF
      GO (m.nRecNo) IN ClassLists
   ENDIF

   This.LoadTree()

OTHERWISE
   * Nothing to do
ENDCASE

RETURN
```

Make sure to SET EXACT ON in the form. Some of the sftreeviewcontainer code expects that setting.

Drag-and-drop issues

While both forms of drag-and-drop are powerful and offer a lot, there are also some issues in working with them.

Drag from text controls

The first issue is the interaction between entering data and starting a drag in text controls. (Text controls here means any control that lets you type text in, specifically textboxes, editboxes, spinners and comboboxes.) There are several variations of this problem, depending on which type of drag-and-drop you're using and how you've configured it.

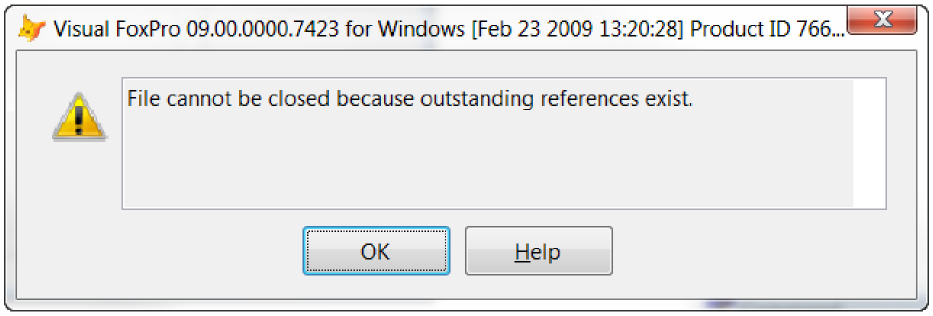
With native drag-and-drop, when a text control is set for automatic dragging (DragMode = 1), you can't click into the control to type. To enter data, you have to tab into it. The best solution to this problem is stay away from automatic dragging; with manual dragging started based on MouseDown, there's no problem clicking into a text control.

For OLE drag-and-drop, the problem is a little more subtle. There, you're not dragging the whole control, but the highlighted text from the control. That means there are two discrete steps to starting a drag: highlighting some text and dragging it. To do that with the mouse, you have to first highlight and release the mouse button, and then push the button again to start the drag. This one is a training issue, since there's no obvious work-around.

Form close problems

An example in "Putting OLE Drag-and-drop together," earlier in this paper, demonstrates changing the caption of a button by dragging text onto it. The first version of that example used a checkbox rather than a button as the drop target; while the code worked, after the drop, the form wouldn't close.

The failure was silent; clicking the Close button (which issues ThisForm.Release) or the form's built-in close button, or choosing Close from the window's menu all did nothing. The only clue to what was going on came from clicking the Modify Form button in the Standard toolbar. When I did that, the message in **Figure 19** appeared. The same problem occurred if the drop target was an option button. To close the form, I needed to issue CLEAR ALL.


Visual FoxPro 09.00.0000.7423 for Windows [Feb 23 2009 13:20:28] Product ID 766...

File cannot be closed because outstanding references exist.

Figure 19. After dropping text on checkboxes or option buttons, this error prevents closing the form.

The materials for this session contain an example form (OLESampleNoClose.SCX) that demonstrates the problem. The only work-around I've found is to not drop text onto those controls with OLE drag-and-drop.

Debugging

Debugging drag-and-drop code is tricky. You can't suspend during a drag-and-drop operation. With native drag-and-drop, if an error occurs before the DragDrop method, the drag is cancelled. With OLE drag-and-drop, you can't use the mouse to navigate in the error dialog.

The inability to suspend, of course, means that you can't step through drag-and-drop code. That means you have to use techniques such as logging. This is actually one of the reasons I created the example classes included in this session's materials. While working on this session, I frequently found myself adding information to the classes, as well as using the form's messaging mechanism to help debug specific issues. As the classes show, DEBUGOUT works during drag-and-drop; so does the Event Tracker. Use those two techniques to understand what's going on in a particular drag-and-drop sequence.

Final thoughts

While a conventional data entry application may have no need for drag-and-drop, there are plenty of applications that can benefit from it, and users expect to be able to do certain tasks, like moving files around, by dragging and dropping.

Fortunately, VFP is up to the task. While drag-and-drop has a lot of moving parts, the power it offers makes it worth learning how it all works. To simplify the learning task, I recommend ignoring native drag-and-drop and putting your efforts into OLE drag-and-drop, which can do everything the native version can and more.