Using Visual FoxPro's ActiveX Controls

Doug Hennig

Partner

Stonefield Systems Group Inc.

1112 Winnipeg Street, Suite 200

Regina, SK Canada S4R 1J6

Phone: (306) 586-3341

Fax: (306) 586-5080

Email: dhennig@stonefield.com

World Wide Web: www.stonefield.com

Introduction

Visual FoxPro includes a lot of ActiveX (formerly OLE) controls that can add great new features to your applications. Many of these, including the TreeView, ListView, StatusBar, and CommonDialog controls, allow you to create a Windows 9x interface for your applications. Others, such as the MAPI controls and Comm control, allow you to add messaging and serial port control to your applications. Some of these controls are less useful: the "Threed" controls, for example, provide very little beyond what the native VFP controls already have, and seem to just be there for Visual Basic or Access usage.

The ActiveX controls are easy to get to: just select *ActiveX Controls* from the *View Classes* menu in the Form Control toolbar, and 30 new controls appear in the toolbar. Simply drag a control from the toolbar as you would any native control, give it a name, set some properties, and away you go. In addition to the VFP Properties sheet, each control has a custom Properties sheet. To access it, right-click on the control and select the appropriate item from the menu that appears.

To make things even better, you can take advantage of the fact that VFP is the only development environment that allows you to subclass ActiveX controls to add additional behavior as necessary. For example, you could create a wrapper class for a control to hide complexity and simplify usage, or add custom properties and methods to further extend the controls. I've done just that with a few of the controls, and provided the source code for these subclasses on the conference disk.

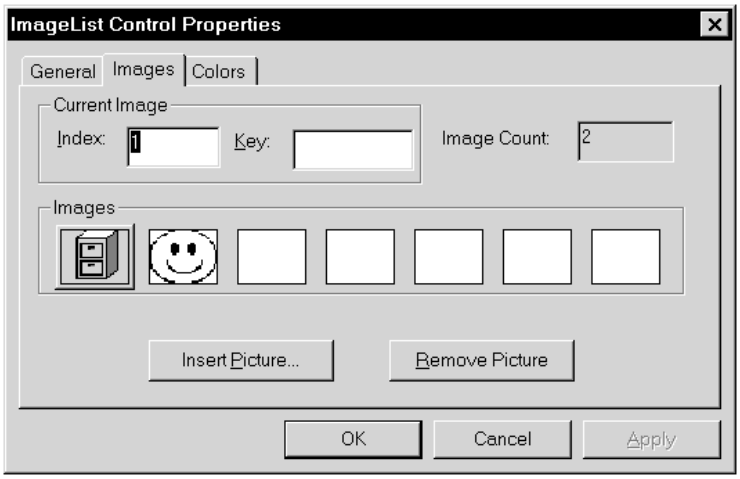
Help files are included to document the properties, events, and methods (PEMs) of the controls. There are three help files: \WINDOWS\SYSTEM\MSACAL70.HLP (for the Calendar control), \WINDOWS\SYSTEM\SYSINFO.HLP (for the SysInfo control), and \VFP5\CTRLHELP\CTRLREF.HLP (for the rest of the controls). In general, you can simply press F1 when an ActiveX control is selected in the Form or Class Designer to bring up the help topic for that control (I'll note some exceptions to this later). However, the documentation for these new controls is not great for a VFP programmer: the text and examples are oriented toward Visual Basic or Access, the organization for some of the topics is weird, and there are tons of errors and omissions. I'll address these concerns in the discussion for each control.

Since there are a lot of new controls and only a limited amount of time for this session, I can't possibly cover them all. I chose the controls I thought would be used the most by the majority of the audience: ImageList, TreeView, Common Dialogs, Calendar, Progress Bar, and Slider. This session focuses on how to use these controls in VFP, gives example of when they're useful, and discusses the most commonly used PEMs so you can customize their appearance or behavior.

ImageList Control

The ImageList control is very simple, but we'll discuss it first because other controls we'll discuss use it. The ImageList control preloads a number of images so other controls (such as the TreeView and ListView controls) can have a source of images. Other than that, it doesn't do anything useful itself.

While this control has a few properties, events, and methods, you're not likely to use any of them. Most likely, you'll use the ImageList Control Property sheet (available from the right-mouse menu) to load the images you'll need for the other controls and set the image sizes and colors.



To load images into an ImageList control, simply drop one from the ActiveX controls toolbar onto a form, name it, bring up the ImageList Control Properties sheet, choose the *Images* page, and insert the pictures you need. The index value shown for each picture (starting from 1) is used in other controls to select an image. For example, the TreeView control, which we'll see in a moment, has an Add method to add a new node, and uses the index number of the associated ImageList control as the image to use for the node.

The ImageList control is located in COMCTL32.OCX (in \WINDOWS\SYSTEM) and is documented in CTRLREF.HLP.

TreeView Control

You've already used the TreeView control—it's the control used in the left pane of the Windows Explorer and is used in the VFP Project Manager and Class Browser. This control is visually more appealing that the Outline control available in VFP 3, and is much easier to work with. For example, the nodes in the Outline control had to be added to the control in the order they appear in outline. Nodes can be added to the TreeView control in any order because you can specify which node is the parent when you add a new one.

Some uses of this control are:

- Displaying a bill of materials list for an inventory control system.
- Drill downs: for example, displaying a list of customers and drilling down to orders and products for each order. Another example is a list of regions, which expands to the salespeople under each region, each of which expand to their customers, and so on.
- Organization charts: divisions are expanded to branches, which are expanded to supervisors, which are expanded to employees.
- Any other hierarchical display of information.

The TreeView control is located in COMCTL32.OCX (in \WINDOWS\SYSTEM) and is documented in CTRLREF.HLP. The help information on the TreeView control is probably the poorest of all the ActiveX controls: its organization is confusing, the information is wrong in several places, and it's frequently as clear as mud, especially in documenting how some methods and properties are accessed.

One of the main things to understand with the TreeView control is that there are three different types of objects you work with: the TreeView control itself, the Nodes collection, and Node objects. The Nodes collection is like the Controls collection of a form; it enables you to access individual node objects by an index number. However, you can also access individual node objects without going through the Nodes collection. For example, the TreeView SelectedItem property is an object reference to the selected node, and some methods, such as NodeClick and Expanded, accept an object reference to a node as a parameter.

We'll look at the PEMs of these three types of objects separately.

TreeView Control Methods and Events

The TreeView control responds to some of the methods and events we associate with other VFP controls: Click, DblClick, Drag, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus, MouseDown, MouseMove, MouseUp, Move, Refresh, SetFocus, ShowWhatsThis, and ZOrder. One big exception: there's no RightClick event. We'll see how to handle that later.

In addition to these methods and events, the TreeView control has some of its own specific methods and events (note the help file lists Clear and Remove methods, but these really belong to the Nodes collection, not the TreeView control itself):

- *BeforeLabelEdit* and *AfterLabelEdit*: occur before and after the label is edited by the user (like the Windows Explorer, you can click on the selected node and edit its text, although you can disable this automatic editing capability as we'll see in a moment). Code in this event would usually be used to save the new text somewhere, such as a field in a table.
- *Collapse* and *Expand*: fired when the user collapses or expands a node. They receive as a parameter an object reference to the selected node. Note that collapsing or expanding a

node doesn't make it the active node, which is usually confusing to the user. Add the following code to the Collapse and Expand events to ensure the node becomes active (note the call to NodeClick(), a method we'll look at later, is only necessary if you have some custom code in that method):

```
lparameters node
Node.Selected = .T.
This.NodeClick(Node)
```

- *GetVisibleCount*: the number of complete nodes visible in the control. This number might seem to be too low by one if the control can show the majority of the last node.
- *HitTest*: returns a reference to a node object if one exists at the passed X and Y coordinates, or .NULL. if there is no node at those coordinates. One complication: HitTest expects the X and Y coordinates in a unit known as "twips" (used in Visual Basic), while VFP uses pixels. The following code will convert pixels to twips (thanks to Chin Bae and Mark Giesen for figuring this out). The TREEVIEW sample form has this code in its Init method, and has two custom properties to store the calculated values: nTreeFactorX and nTreeFactorY.

```
local liHWnd, ;
   liHDC, ;
   liPixelsPerInchX, ;
   liPixelsPerInchY

* Define some constants.

#define cnLOG_PIXELS_X      88
   * From WINGDI.H
#define cnLOG_PIXELS_Y      90
   * From WINGDI.H
#define cnTWIPS_PER_INCH  1440
   * 1440 twips per inch

* Declare some Windows API functions.

declare integer GetActiveWindow in WIN32API
declare integer GetDC           in WIN32API ;
   integer iHDC
declare integer GetDeviceCaps   in WIN32API ;
   integer iHDC, integer iIndex

* Get a device context for VFP.

liHWnd = GetActiveWindow()
liHDC  = GetDC(liHWnd)

* Get the pixels per inch.

liPixelsPerInchX = GetDeviceCaps(liHDC, cnLOG_PIXELS_X)
liPixelsPerInchY = GetDeviceCaps(liHDC, cnLOG_PIXELS_Y)

* Get the twips per pixel and store in properties of the form.

with This
```

```
   .nTreeFactorX = cnTWIPS_PER_INCH/liPixelsPerInchX
   .nTreeFactorY = cnTWIPS_PER_INCH/liPixelsPerInchY
endwith
```

A couple of other things are required to get this to work:

```
_VFP.AutoYield = .F.
sys(2333, 0)
```

Another complication: the X and Y values passed to the various TreeView methods vary depending on the method. "ActiveX Control Event" methods (those that have this comment automatically added to the code, such as MouseDown and MouseMove) get X and Y values relative to the left and top edges of the TreeView control, and HitTest works fine with these. VFP container methods (those that don't have this comment, such as DragDrop and DragOver) get X and Y values relative to the form. Thus, in these methods, you need to subtract the Left and Top values of the TreeView control before using them with HitTest().

HitTest() can help solve an issue with right-mouse clicks. You might want to display a popup menu when the user right-clicks on a node but right-clicking on a node doesn't make it the selected node. The following code in the MouseDown event of the control can be used to handle this (this is the only way you can handle right-mouse clicks since there is no RightClick event). Again, the call to NodeClick() is only necessary if you have some code in that method.

```
parameters Button, Shift, X, Y
local loNode
if Button = 2

* If this is the right mouse button, get a reference
* to the node under the mouse.

   loNode = This.HitTest(X * Thisform.nTreeFactorX, ;
      Y * Thisform.nTreeFactorY)

* If we have a valid node, select it.

   if not isnull(loNode) and loNode.Key <> This.SelectedItem.Key
      loNode.Selected = .T.
      This.NodeClick(loNode)
   endif not isnull(loNode) ...
   * show the right-click menu now
else
   * handle any left-mouse click stuff necessary
endif Button = 2
```

- *NodeClick*: fired when the user clicks on a node (before the Click event). NodeClick receives as a parameter an object reference to the selected node object. Typically, this method is used to update some things (such as the values of other controls) when a node is selected. If the code in NodeClick takes too long to execute, the selected item will be highlighted but the former item will have a dashed line around it. Moving the mouse (even without clicking the mouse button) restores the highlight to the former item. To avoid this problem, you can add the following code to the NodeClick event to ensure the node clicked on becomes selected:

```
Node.Selected = .T.
```

- *StartLabelEdit*: used to manually invoke the editing of a node's label. This is used when the LabelEdit property is 1-Manual.

TreeView Control Properties

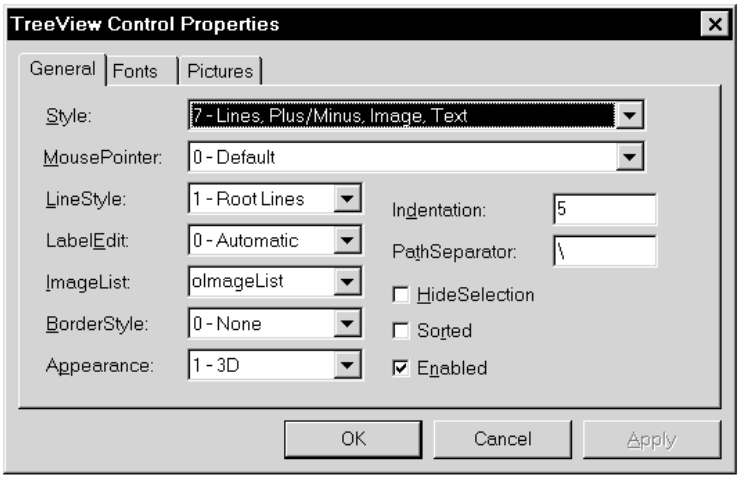
As with events, TreeView controls support some of the same properties other controls do, including DragIcon, DragMode, Enabled, and Visible. Many of the TreeView-specific properties can be set in the TreeView Control Property sheet invoked from the right-mouse menu. These include Style, LineStyle, Indentation, PathSeparator, and HideSelection.

Of the properties which can be edited in the property sheet, the ones you're most likely to change from the defaults are:

- *Style*: indicates what appears in the TreeView control. You can choose whether images appear or not, whether lines are displayed, whether the plus/minus marks are shown, etc.
- *LineStyle*: indicates whether "root lines" are displayed or not. If you don't set this to 1-Root Lines, you won't get the plus/minus mark for the top level objects, regardless of how Style is set.
- *LabelEdit*: if you don't want the user to be able to edit the text for each node, set this to 1-Manual.
- *ImageList*: a reference to an ImageList control containing the images used by the nodes in the control; see the ImageList section in this document for information on this simple control. Unfortunately, this property can't be set visually; you have to do it programmatically using code something like the following in the Init method of the form:

  ```
  This.oTree.ImageList = This.oImageList.Object
  ```

  Note that you need to do this in the Init of the form rather than the TreeView because the TreeView might instantiate before the ImageList, in which case trying to set the ImageList property to a not-yet-existent object would fail. Also, note the "Object" keyword in code above; this is required.
- *Indentation*: how much each child node is indented by.



- *HideSelection*: if you don't turn this option off, the selected node doesn't stay highlighted when the TreeView control loses focus. This can be very confusing to the user.
- *Font*: the font name, size, and style used for the node text.

The properties which aren't available from the property sheet are (note the help file includes a ScrollBars property, but there is no such property):

- *DropHighlight*: this is supposed to be used in the DragOver event of the TreeView control so you can highlight a node as the mouse passes over it. However, this only works in VFP 6 with version 6 of the TreeView control; setting this property to a node object in an older control gives a "type mismatch" error. In that case, you can get around this problem by using HitTest() to determine which node the mouse is over and setting that node's Selected property to .T. The TREEVIEW sample form uses this technique in its DragOver event.
- *Nodes*: references the Nodes collection in the control.
- *SelectedItem*: a reference to the currently selected node object.

Nodes Collection Methods

The following are the methods of the Nodes collection of the TreeView control (referenced by specifying <Object>.Nodes, where <Object> is the name of the TreeView control):

- *Add*: adds a new node and returns a reference to it. It uses the following syntax:

```
<Object>.Nodes.Add(Relative, Relationship, Key, Text, Image, ;
   SelectedImage)
```

  where:

  - *Relative*: the index or key of an existing node object. If it isn't specified, the new node is placed at the end of the top node hierarchy.
  - *Relationship*: where the new node should be placed relative to the node specified in the first parameter:
    - 1: the node is placed at the end of all other nodes at the same level of the relative node.
    - 2: the node is placed after the relative node.
    - 3: the node is placed before the relative node.
    - 4: the node becomes a child of the relative node.
  - *Key*: a unique string used to identify the node. If the tree is loaded with records from a table, I use the primary key of the record (converted to a string if necessary). Otherwise, I just use a sequential number converted to a string.
  - *Text*: the text displayed in the control for the node.
  - *Image*: the index of an image in the associated ImageList control.
  - *SelectedImage*: the index of an image in the associated ImageList control that's shown when the node is selected.
- *Clear*: nukes all nodes.

- *Remove*: removes the node corresponding to the specified index.

Nodes Collection Properties

The following are the properties of the Nodes collection:

- *Count*: the number of nodes.
- *[<Index>]*: an object reference to node number <Index>.

Node Objects Methods

The following are the methods of node objects:

- *CreateDragImage*: this can't be used in VFP because DragIcon expects a CUR file name while this method returns an image.
- *EnsureVisible*: ensure the specified node is visible. This methods scrolls the TreeView control if necessary and expands all parent nodes of the specified node.

Node Objects Properties

The following are the properties of node objects:

- *Children*: .T. if the node object has any child node objects.
- *Expanded*: .T. if the node object is expanded.
- *FullPath*: the concatenation of the Text values of all parent objects leading to this one, separated by the control's PathSeparator property. It's very similar to the concept of a fully qualified file path and name.
- *Image*, *ExpandedImage*, and *SelectedImage*: the appropriate image number in the associated ImageList control.
- *Index*: the index of the node object in the Nodes collection.
- *Key*: the unique key assigned when the node was added.
- *Child*, *FirstSibling*, *LastSibling*, *Previous*, *Parent*, *Next*, and *Root*: point to the appropriate node object relative to the specified one.
- *Selected*: .T. if the node object is selected. Setting this property to .T. automatically highlights the node and sets the Selected property of the previously selected node to .F.
- *Text*: the text that appears in the control.

Loading the TreeView Control

Initially loading the tree, while relatively simple, can have some interesting twists. First, let's look at a straightforward example. Assume we want to load a tree with customers and orders for those customers (this is what the TREEVIEW sample form does). Here's some code that'll do this; this code would likely get called during form initialization (for example, in a custom method of the form called in the Init() method of the form or the Init() method of the TreeView control):

```
with This
```

```
* Set up the CUSTOMER and ORDERS tables.

   select CUSTOMER
   set order to CUST_ID in ORDERS
   set order to COMPANY
   scan

* Add a node for the customer to the tree.

      lcCustomerKey = 'C' + CUST_ID
      .oTree.Nodes.Add(, 1, lcCustomerKey, trim(COMPANY), 1)

* Find the first order for this customer and process each one.

      select ORDERS
      seek CUSTOMER.CUST_ID
      scan while CUST_ID = CUSTOMER.CUST_ID
         .oTree.Nodes.Add(lcCustomerKey, 4, 'O' + ORDER_ID, ;
            dtoc(ORDER_DATE) + ' ' + ;
            transform(ORDER_AMT, '$99,999.99'), 2)
      endscan while CUST_ID = CUSTOMER.CUST_ID
      select CUSTOMER
   endscan
endwith
```

Note the key assigned to each node is the key for the record plus a prefix to indicate what table the record comes from ("C" for CUSTOMER and "O" for ORDER). The prefix avoids the problems where an order and customer have the same primary key, and the combination of prefix and primary key allow us to quickly find the record matching a particular node (using the node's Key property, get the table from the first character, and the primary key to seek from the rest of the string).

The only downside for this code is that it must process every order for every customer before the form can be displayed. If there are a lot of customers or orders, this can take a long time. A better approach is to load only the top level items (customers in this case), then in the Expand() method, check if the node about to be expanded has been loaded with child nodes. If not, load the child nodes just for the current node. This is much faster than loading the entire tree at the beginning. One tip: if you don't load any children for a node, no + will appear for the node. You'll need to add at least one node, even a "dummy" one, for each top level node. When the node is expanded for the first time, remove the dummy node, then add the "real" child nodes.

The TREEVIEW sample form shows how to use this technique. It has an lLoadChildren property that determines whether all orders and products are loaded before the form is displayed or not. Try setting this to .F. and notice how long it takes before the form is displayed.

You might need to reload the tree at some point. For example, if other application users are adding customers or orders, you might need to refresh the tree from time to time with any new customers or orders. To prevent the hassles of figuring out which records are already in the tree and which aren't, you could use the Nodes collection Clear() method to clear the tree, then reload it from scratch. The only problem with this approach is that some nodes may have been expanded. Clearing and reloading the tree causes all nodes to start in the default collapsed view. To overcome this, you can save the Expanded and either Key or FullPath properties of each existing node in an array or cursor, clear and reload the tree, then go through the array or cursor and reset the Expanded property of those nodes that existed before to the saved former value. The GetExpanded and SetExpanded methods of the TREEVIEW sample form do just that.

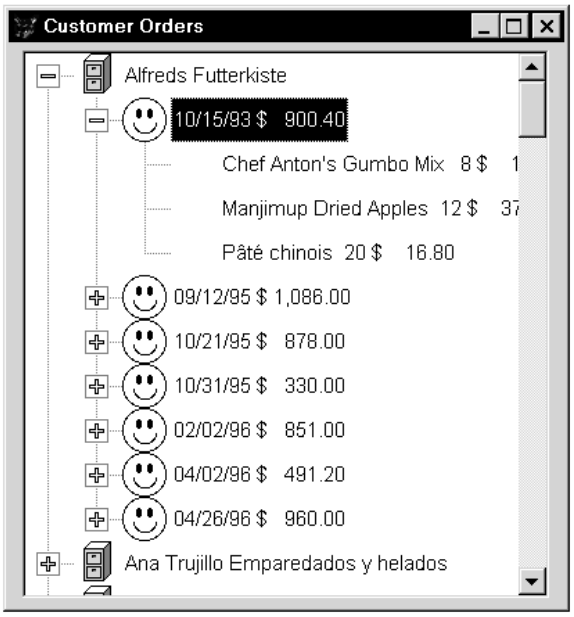
Drag and Drop

Based on messages I've seen on CompuServe, there's a lot of confusion about whether you can do drag and drop with the TreeView control. Some folks think that because VFP doesn't implement drag and drop with other applications such as Word (which is true), it can't do it with ActiveX controls (which isn't true). The reason you can do drag and drop with ActiveX controls even though they're non-VFP applications is that when you drop an ActiveX control on a form, VFP places it inside a VFP container, which of course does support drag and drop. As a result, doing drag and drop with a TreeView control (either from or to the control) works no differently than it does for any other control.

The TREEVIEW sample form implements drag and drop both from and to the control. You can drag a node from the tree to the textbox under the "Dragged Node" label; the text of the node will appear in the textbox. You can also drag the "Drag me to tree" label and drop it on any node in the tree; a new node will be added showing the text of the node you dropped the label on along with the date and time.

Example

The TREEVIEW form included with the source code for this session displays customers, orders for each customer, and the products for each order from the VFP TESTDATA database. Note the ImageList control on the form to provide images for the nodes. This form incorporates most of the tips covered in this session.



Common Dialogs Control

If you've used the FoxPro GETFILE() and PUTFILE() functions, you've probably found that they leave something to be desired:

- There's no way to change the title of the dialog that appears.
- The PUTFILE() function always asks the user if they want to overwrite an existing file.
- In both cases, the specified path must exist or an error message is displayed.

In order to get more flexibility, you need to use the Common Dialogs control that comes with VFP 5 (in COMDLG32.OCX in \WINDOWS\SYSTEM). This control is called "common dialogs" because it can display File, Color, Font, and Print dialogs. All of these dialogs provide more flexibility than the VFP counterparts. For example, while the GETFONT() and GETCOLOR() functions work well in VFP, you have no control over things like whether non-TrueType fonts are available or whether the user can define custom colors. The Font and Color dialogs available from the Common Dialogs control do have these capabilities.

Due to time constraints, we'll just concentrate on the File dialog in this session. If you want more information on the Color, Font, and Print dialogs, see the ActiveX controls help file. Note the Common Dialogs control doesn't appear in the Contents page of the help file, but can be found by pressing F1 when the control is selected or looking for "Common Dialog" in the help index.

Thanks to Ted Roche for helping me understand the multiple file selection issues of this control.

Methods

The Common Dialogs control has five methods, none of which accept parameters:

- ShowOpen() shows an Open File dialog
- ShowSave() shows a Save File dialog
- ShowPrinter() shows a Printer dialog
- ShowFont() shows a Font dialog
- ShowColor() shows a Color dialog

We'll use only the first two methods in this session.

Properties

The following are the most frequently used properties of the Common Dialogs control that pertain to the File dialog (some of the properties are also used by the Font, Color, and Printer dialogs); see the help file for information on other lesser-used properties.

- *CancelError*: set this property to .T. if you want an error generated (error #1429) if the user chooses Cancel. You can trap the error in the Error method of the control and handle it gracefully. This is necessary because while the equivalent VFP functions return a special value (usually blank) to indicate that the user chose Cancel, these dialogs do not.

- *DefaultExt*: this extension is automatically appended to the filename the user enters if it doesn't have an extension.
- *DialogTitle*: the title of the dialog window.
- *FileName*: used as the initial value for the filename, plus it contains the value entered by the user upon return from the dialog. If multiple file selection isn't allowed, or it is allowed but the user only selects a single file, FileName contains the complete path and filename for the selected file. If multiple file selection is allowed and the user selects more than one file, FileName contains the path for the files selected followed by a separator followed by each filename (without a path) separated by a separator. The separator is CHR(0) in Windows 95 and NT 4 or a space in Windows NT 3.51. For example, if the user chose three tables from the sample data directory that comes with VFP, FileName might contain "C:\VFP5\SAMPLES\DATA <separator> CUSTOMER.DBF <separator> ORDERS.DBF <separator> ORDITEMS.DBF". See the SetProperties() method of the SFFileDialog class (described later) for some code to parse FileName into the path and individual filenames.
- *FileTitle*: the name of the selected file without a path if multiple file selection isn't allowed or blank if it is.
- *Filter*: the specification of acceptable files. There are two parts to the filter: the description the user sees (for example, "Database Files") and the extension of those files (such as "*.DBC"). The description and specification are separated by the vertical line character (|). You can provide more than one filter by separating description-specification sets with the vertical line character. For example, "Database Files | *.DBC | Tables | *.DBF | All Files | *.*".
- *FilterIndex*: the default filter to use. The first one in the Filter property is 1.
- *Flags*: this property controls the appearance and the behavior of the dialog. See below for the values that can be used.
- *HelpFile*: the name of the help file used when the user clicks on the Help button in the dialog.
- *HelpCommand*: the most common settings for this property (see the help for this control for other settings) are 0x1 (use a HelpContextID), 0x101 (use a key word), or 0x105 (use a partial key word). Note: 0x notation, new in VFP 5, indicates a hexadecimal value.
- *HelpContext*: the HelpContextID for the topic in HelpFile to be displayed if HelpCommand is set to 0x1.
- *HelpKey*: the key word to find in the help file if HelpCommand is set to 0x101 or 0x105.
- *InitDir*: the initial directory for the dialog.

Note the help mentions two other properties, Path and Drive, but these properties do not in fact exist.

The Flags property acts similar to the DialogBoxType parameter in the VFP MESSAGEBOX() function; you add various values together to obtain the appearance and behavior you want. Unfortunately, even if the flag has the same meaning for more than one type of dialog, it might not have the same value. For example, the Show Help Button flag is 0x8 for the Color dialog and 0x10 for the File dialog. Because the values for each setting are rather arcane, I created an include file called COMMDLG.H file (included on the source code disk) that defines constants for each value. To set the Flags property of the control,

simply add the values of the settings you want together. For example, to display the help button and use an overwrite prompt, you'd do something like:

```
This.oCommonDialog.Flags = cnFILEDLG_OVERWRITE + cnFILEDLG_SHOWHELP
```

Here are the most common choices for Flags (see the help file for others):

- *Allow Multiple File Selection* (0x200; cnFILEDLG_MULTIPLE in COMMDLG.H): allows selection of multiple files. If this option is turned on, the dialog box appears quite different than the usual Windows 95 dialog unless you also add 0x80000 (cnFILEDLG_EXPLORER) to Flags; note the help file incorrectly specifies this setting as 0x8000.
- *Prompt if File Doesn't Exist* (0x2000; cnFILEDLG_PROMPTNEW): prompts the user to create a file that doesn't currently exist. If you turn this option on, you don't need to turn on the "Path Must Exist" and "File Must Exist" flags, since they get set automatically ("Path Must Exist" is turned on and "File Must Exist" is turned off). This flag isn't used with the Save dialog.
- *File Must Exist* (0x1000; cnFILEDLG_FILEEXIST): displays an error message if the user enters the name of a file that doesn't exist. If you turn this option on, the "Path Must Exist" flag is automatically turned on. This flag isn't used with the Save dialog.
- *Path Must Exist* (0x800; cnFILEDLG_PATHEXIST): displays an error message if the user enters a path that doesn't exist. VFP's GETFILE() and PUTFILE() functions have this permanently turned on, which is one reason why you might want to use the Common Dialogs control instead.
- *Hide Read Only* (0x4; cnFILEDLG_HIDERO): hides the Read Only check box in the dialog.
- *Help Button* (0x10; cnFILEDLG_SHOWHELP): displays the Help button in the dialog.
- *Overwrite Prompt* (0x2; cnFILEDLG_OVERWRITE): displays an "overwrite this file" message if the user enters the name of an existing file. This flag is only used with the Save dialog. VFP's PUTFILE() function has this permanently turned on, which is one reason why you might want to use the Common Dialogs control instead.
- *Share Aware* (0x4000; cnFILEDLG_SHAREAWARE): allows the user to select a file that's already open in some application.
- *Don't Change Directory* (0x8; no constant defined): the File dialog does something you might not expect: if the user chooses a different directory than the current one, that directory becomes the current directory upon exit from the dialog. This setting prevents that behavior.

Subclassing Common Dialogs

To make it easier to use the Common Dialogs control, I created an abstract subclass of the control called SFCommonDialog (in the ACTIVEX.VCX included on the source code disk). SFCommonDialog has two new custom properties: lShowHelpButton (set it to .T. to display a help button) and lCancelled (set to .T. if the user cancels the dialog). It also has a new custom method (SetFlags) that sets some properties so we can detect if the user cancelled the dialog. The Error method handles the case where the user cancelled the dialog (CancelError is automatically set to .T. so an error is triggered if the user cancels) by setting the custom property lCancelled to .T.

I created a subclass of SFCommonDialog called SFFileDialog used specifically for File dialogs. SFFileDialog uses COMMDLG.H as its include file. It has several custom properties representing the various options for the Flags property so you don't have to remember what constants to use for which settings. These properties, which are all set to .F. by default, are lAllowMultiple, lFileMustExist, lOverWritePrompt, lPathMustExist, lPromptNew, lShareAware, and lShowReadOnly. Another custom property, cPath, is set to the path of the filename the user enters. cOldCurrDir is a protected custom property used to hold the current directory before the dialog is called. A custom array property, aFiles, is used to hold the filenames (without paths) of the files selected by the user. Two custom methods, OpenFile() and SaveFile(), display the appropriate dialog. Both of these methods call the custom methods SaveDirectory() and RestoreDirectory() to save and restore the current directory, SetFlags() to set the Flags property to the appropriate value based on the settings of the custom properties mentioned earlier, and SetProperties() to set the cPath, lCancelled, and aFiles properties appropriately.

OK, now that we've built it, what's SFFileDialog good for? I use it whenever I need to ask the user for a file because it gives me more control over the appearance and behavior of the dialog that PUTFILE() or GETFILE(). For example, say you have an import function in an application. If the user has three files they want to import, do you really want to make them call the function up three times? It'd be a lot easier if they could just select multiple files and let the import function import them all in one sitting, especially if the process takes some time to run.

Here's how you'd do something like that. Drop an SFFileDialog object on a form by dragging it from the Project Manager to the form. SFFileDialog is a non-visual control so it's not visible on the form at runtime (if it seems strange that an object that displays a dialog is non-visual, remember that the dialog isn't displayed until a method of the control is called). Set the various properties of the control as desired. For example, to allow the user to select multiple files, set the lAllowMultiple property to .T. The Click() method of an Import button on the form would call the OpenFile() method of the SFFileDialog object to display an Open file dialog. Upon return from OpenFile(), it would check the lCancelled property to determine if the user cancelled the dialog. If not, the code would use the aFiles (which contains the names of the file selected) and cPath (which contains the directory the files are in) properties to import the files. Here's what the code might look like:

```
with Thisform.oFileDialog
   .OpenFile()
   if not .lCancelled
      for lnI = 1 to alen(.aFiles)
         lcFile = .cPath + .aFiles[lnI]
         Thisform.ImportFile(lcFile)
      next lnI
   endif not .lCancelled
endwith
```

The COMMDLG form on the source code disk is a "testbench" for the Open and Save dialogs. It includes an SFFileDialog object, and many of the controls in the form have properties of this object for their ControlSource. You can set various flags and other properties and click on the Test button to see what affect they have on the appearance and behavior of the dialogs.

Other Notes

VFP's GETFILE() and PUTFILE() functions have some functionality the Common Dialogs control doesn't:

- You can specify the text to display instead of "File name" beside the file name. However, this doesn't help a lot since the text to display can only be a very short phrase (anything longer than about 10 characters is truncated).
- GETFILE() allows you to specify the prompt on the Open button and indicate whether a New or None button is displayed.
- As I mentioned earlier, GETFILE() and PUTFILE() return a blank filename if the user chooses Cancel while the File dialog does not. You can work around this by setting the CancelError property to .T. and trapping error #1429 (typically setting the FileName and FileTitle properties to blank).

Calendar Control

The Calendar control is located in MSACAL70.OCX (in \WINDOWS\SYSTEM; the help file erroneously mentions MSACAL.OCX). The help file is called MSACAL70.HLP. This control provides you with the ability to include a calendar in your applications. An obvious use of this control is to display a calendar when the user right-clicks on a date field.

Methods and Events

The methods in the Calendar control are mostly for programmatic date manipulation, including NextDay, NextWeek, NextMonth, PreviousDay, PreviousWeek, and PreviousMonth. You could, of course, provide buttons or other means to call these methods, but since the user can easily change the date by clicking on various controls in the calendar, I don't see a lot of need for it.

In addition to the usual events such as Click, DblClick, and KeyPress, the Calendar control has AfterUpdate, BeforeUpdate, NewMonth, and NewYear events that allow you to do some special processing when the user changes something if necessary. I think the most frequent event you'll use is DblClick, which could be used to release or hide the Calendar control when the user selects a date.

Properties

The properties of the Calendar control are of more interest than the methods and events. You'll probably want to set up some of the properties, such as colors (BackColor, DayFontColor, GridFontColor, GridLinesColor, TitleFontColor), fonts (DayFont, GridFont, and TitleFont), and other properties defining the appearance of the control (DayLength, FirstDay, GridCellEffect, MonthLength, ShowDateSelectors, ShowDays, ShowHorizontalGrid, ShowTitle, and ShowVerticalGrid), in the Calendar Control Properties sheet available from the right-click menu. The Value property contains the date selected in the calendar and the Day, Month, and Year properties contain the appropriate parts of the date. Usually, you'll want to set the Value of the control to a specific date (for example, in the Init of the control) so that date is highlighted by default, and get the Value after the user has chosen a date.

Examples

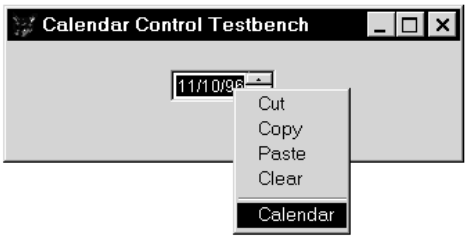
I've created a container class that contains a Calendar control and several buttons (SFCalendar in ACTIVEX.VCX on the source code disk) and an SFDateSpinner class (in CONTROLS.VCX on the source code disk) that instantiates SFCalendar when the user chooses "Calendar" from a right-click menu.

The SFDateSpinner class is a container class with a TextBox and a Spinner (with just the up and down arrows visible). The spinner increments and decrements the date in the TextBox, and the TextBox has code in its KeyPress method to simulate Quicken date keystrokes. When the user right-clicks on the TextBox, the ShortcutMenu method is called to display a shortcut menu (hard-coded in this method to

encapsulate the control). If the user chooses “Calendar” from this menu, the ShowCalendar method is called. This method instantiates an object whose class is specified in the cCalendarClass property (the default value is SFCalendar). NEWOBJ.PRG helps with the instantiation, ensuring the class library the class is located in is open.

The SFCalendar control accepts an object parameter so it can change the value of the object to the chosen date before being released. The control is released when the user double-clicks on a date or chooses the Save or Exit buttons in the container.

The CALENDAR form on the source code disk includes an SFDateSpinner object so you can try these classes out.



ProgressBar Control

The ProgressBar control is located in COMCTL32.OCX and is documented in CTRLREF.HLP. This control gives us a Windows 95 progress bar, such as the one displayed when you copy large files from one drive to another. This control should be used whenever a process will take some time and you want to inform the user of the progress. Examples include performing long calculations such as payroll amounts, performing complex queries before printing a report, archiving records, etc. The progress bar can be updated after each record or batch of records, or after each step in a discrete sequence of tasks.

Methods and Events

The ProgressBar control responds to some of the methods and events we associate with other VFP controls: Click, Drag, DragDrop, DragOver, MouseDown, MouseMove, MouseUp, Move, ShowWhatsThis, and ZOrder.

Properties

As with the Calendar control, the properties of the ProgressBar control are more interesting than the methods or events because this is largely a visual control. Many of these affect the appearance of the control and are most easily set at design time in the VFP Property sheet or the ProgressBar Control Properties sheet selected from the right-click menu. These include Align (which determine whether the position of the control can be moved or whether it automatically is attached to the top, bottom, left, or right edge of the form), Appearance (flat or 3D), and BorderStyle.

The properties we're more interested in at run time are Min, Max, and Value. Min and Max represent the range of the control's values, and default to 0 and 100, respectively. The length of the bar in the control is controlled by the Value property.

Examples

ACTIVEX.VCX on the source code disk contains an SFThermometer class. This class is a container class with a progress bar, several labels, and a Cancel button. It has custom methods (SetTitle and SetMaximum) that set the caption for the "main" label and the maximum value for the ProgressBar control. To update the meter, use the Update method. It accepts two parameters: the current value of the meter and the caption for the "current task" label in the form. The value passed to Update is interpreted differently depending on the setting of the lPassPercent property of the form. If lPassPercent is .T., Update expects that the value passed is a percentage; if not, it expects the value is an absolute value and calculates the percentage by dividing by the maximum value.

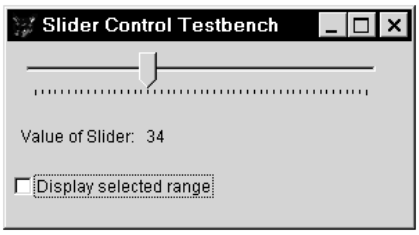
SFThermometer uses a very interesting technique to permit a "hard" loop from being interrupted. The Update method checks to see if the mouse is over the Cancel button and if it's been pressed. If so, it uses the new `doevents` command to allow the event to be processed (that is, the Cancel button gets pressed down), and then sets a flag that the user cancelled the process. The loop which calls the Update method checks this flag to decide if it needs to continue processing or not. I originally used `doevents` by itself

in the Update method to allow the Cancel button to be pressed (rather than checking the mouse position and down state), but this took an *order of magnitude* longer to process!

PROGRESS.PRG is a sample program that shows how SFThermometer can be used. The PROGRESS form simply has an SFThermometer container dropped on it and PROGRESS.PRG runs this form to display the meter.

Slider Control

The Slider control is located in COMCTL32.OCX and is documented in CTRLREF.HLP. A Slider is a control similar to a slider volume control on a stereo; it has a bar representing the range of values for the control and a pointer that can be dragged along the bar to indicate the chosen value. This control is used for entry of numeric values, but is more likely to be used in a "preferences" or "properties" type of dialog rather than for data entry, where a TextBox or Spinner would be more appropriate.



Methods and Events

The Slider control responds to some of the methods and events we associate with other VFP controls: Click, Drag, DragDrop, DragOver, GotFocus, KeyDown, KeyPress, KeyUp, LostFocus, MouseDown, MouseMove, MouseUp, Move, Refresh, SetFocus, ShowWhatsThis, and ZOrder.

The Change event is similar to the InteractiveChange event of other controls; it fires when the Value property changes. The Scroll event fires continuously as the slider is dragged along the bar.

The ClearSel method clears the selected area of the control (see below). GetNumTicks returns the number of ticks in the control.

Properties

Many of the properties of the Slider control affect the appearance of the control and are most easily set at design time in the VFP Property sheet or the Slider Control Properties sheet selected from the right-click menu. These include BorderStyle, LargeChange (the number of ticks the slider will move when you press PgUp or PgDn or when you click the mouse to the left or right of the slider), SmallChange (the number of ticks the slider will move when you press the left or right arrow keys), Orientation (horizontal or vertical), TickStyle (do ticks appear across the top/left edge, bottom/right edge, both or neither edges), and TickFrequency.

The properties we're more interested in at run time are Min, Max, and Value. Min and Max represent the range of the control's values, and default to 0 and 100, respectively. The position of the slider along the control is controlled by the Value property.

The SelectRange property controls whether the slider displays a selected range or not. If SelectRange is set to .T., the SelStart and SelLength properties determine the starting position and length of the selected range.

Examples

The SLIDER form on the source code disk shows an example of the Slider control.