Modernize Your Applications with DBI's ActiveX Controls

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Corporate Web sites: stonefieldquery.com
stonefieldsoftware.com
Personal Web site : DougHennig.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

DBI Technologies is the only ActiveX control vendor that actively supports VFP developers. Their Studio Controls for COM suite provides 88 controls that allow you to modernize the user interface of your applications so they look more like Microsoft Office. This document looks at several of the controls in this suite, showing you how to implement them to freshen and extend the life of your applications.

Introduction

As time goes on, Visual FoxPro's native controls get more and more outdated looking compared to those in modern applications such as Microsoft Office. I've written numerous white papers over the years on how to modernize VFP applications (available from the Technical Papers page of my personal web site, doughennig.com) but one way to do it is to use ActiveX controls that look modern and allow you to customize them as necessary. Unfortunately, most of the ActiveX control vendors have fallen by the wayside or don't specifically develop and test their controls (or document them) with VFP in mind. There is one, however, that has supported the VFP community since day one, since they started as VFP developers themselves: DBI Technologies (https://www.dbi-tech.com).

DBI has a several ActiveX control suites that work with VFP. The one this document focuses on is Studio Controls for COM, https://tinyurl.com/va8mufq. There are both 32-bit and 64-bit versions of this suite. Why would a VFP developer care about 64-bit controls? Because there's a 64-bit version of VFP available called VFP Advanced available at http://www.baiyujia.com/vfpadvanced/. If you're using it to create 64-bit applications, you can't use any 32-bit ActiveX controls, including the ones that come with VFP, so your only option is the DBI control suite. Fortunately, they have replacements for just about every Microsoft control.

DBI actively supports VFP, including code snippets in the documentation (each control has its own CHM file) and VFP samples for every control.

According to DBI's web site, Studio Control for COM comes with 88 controls that provide "modern Windows UI designs, Outlook and Office 365 style appointment scheduling, data entry, navigation, reporting and data presentation controls for custom branded dashboards." Twenty-five of them are enhanced, Unicode versions of other controls, so there are really 63 unique controls. The 64-bit version doesn't have as many; it contains 25 controls, in both 32- and 64-bit versions.

Obviously, I'm not going to discuss 88 (or even 63) controls. Instead, I'll focus on some I'm particularly interested in. I'll look at the enhanced versions of these controls (which all have a "ctx" prefix, such as ctxTreeView) rather than the older equivalents (which all have a "ct" prefix).

ctxTreeView

One of the most important ActiveX control is the TreeView. I use it all over the place: displaying list of reports in folder; displaying databases, their tables, and the fields in those tables; displaying classes inside class libraries, etc. I have written several articles over the years on the TreeView control, including "The Mother of All TreeViews," which discusses a class named SFTreeView that encapsulates all of the quirky behavior of the Microsoft TreeView so you don't have to worry about it.

ctxTreeView is DBI's version of a TreeView control. It has a lot of features that aren't available in Microsoft's version, including:

- Title and header, including control over colors and fonts (1 and 2 in **Figure 1**).
- Columns, including the ability to sort on a column (3 in Figure 1).
- Check boxes and radio buttons (4 in Figure 1). The Microsoft control allows check boxes in front of nodes but the DBI control can put them in any column, and you can configure what they look like.
- Node sub-text, including control over color and font (5 in Figure 1).
- Node headers, which ignore column formatting and have their own independent colors and fonts.
- A built-in image list, eliminating the need for a separate ImageList control.
- ToolTips for nodes.
- Multi-selection of nodes.
- Control over how selected nodes appear, both when the TreeView has focus and when it doesn't.
- Ability to print or create a JPG of the current content of the TreeView.
- Virtual mode which saves memory by not keeping the text for every node in memory but instead getting it on demand.
- Automatic directory loading.
- Ability to programmatically display the TreeView Properties dialog, even within an EXE.
- Ability to persist settings and nodes to XML.

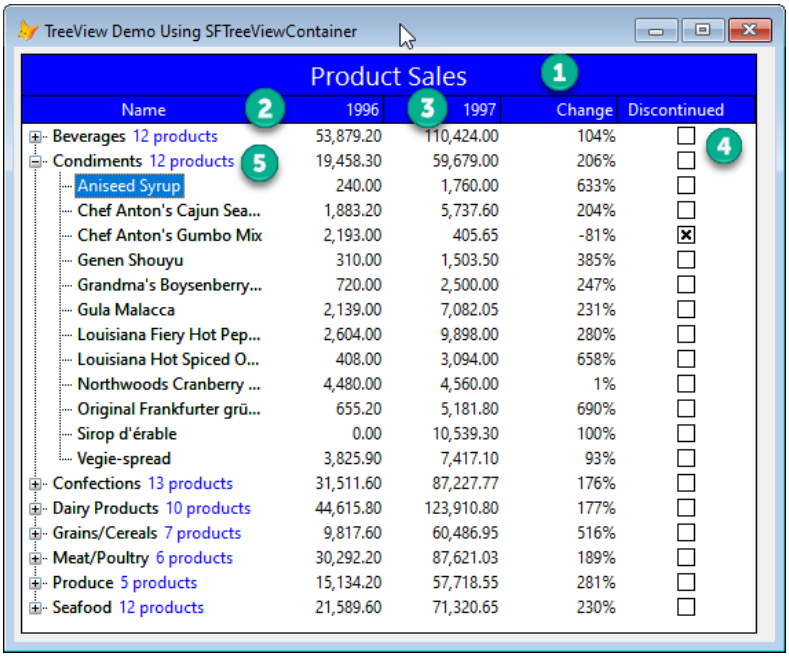

Figure 1. The ctxTreeView control has a lot of features missing from the Microsoft control.

Performance

One of the first things I thought of when looking at the DBI control was how it compares in performance to its Microsoft counterpart. As you may know, loading the Microsoft TreeView can be slow, especially if there are a lot of nodes, so most people use a "lazy loading" mechanism, in which only top-level nodes are loaded and child nodes are only loaded when a parent node is expanded for the first time.

To test the performance of the DBI control versus the Microsoft control, I created a couple of test forms: MSTreeViewPerformanceTest.scx and DBITreeViewPerformanceTest.scx (**Figure 2**).

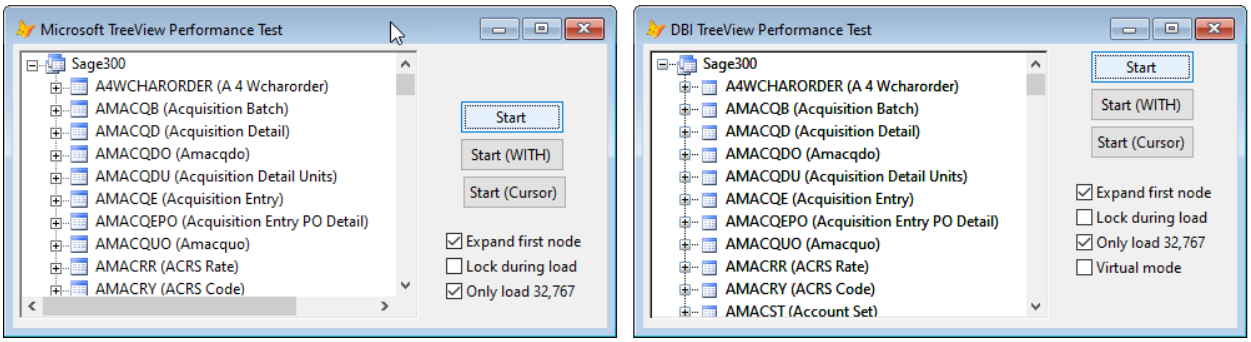

Figure 2. Performance test forms.

The options in these forms are:

- *Start*: loads the TreeView from a table containing the Sage 300 accounting system data dictionary. Under the database node are table nodes and under each table are nodes for the fields in that table.
- *Start (WITH)*: loads the TreeView using statements inside a WITH Thisform.oTree structure to minimize the number of accesses to the members of the TreeView control.
- *Start (Cursor)*: loads the TreeView from a cursor that's sorted in the order the nodes should appear, so add nodes are added to the end of the TreeView rather than adding all the parent nodes and then the child nodes for each parent.
- *Expand first node*: turn this on to expand the database node during loading to see if there's any impact on performance or whether the control flickers as it has to draw the nodes as they're added.
- *Lock during load*: turn this on to use a Windows API call to lock the TreeView control while loading (the equivalent of setting the VFP LockScreen property, which ActiveX controls don't respect, to .T.). Theoretically, this speeds up loading because the TreeView doesn't have to refresh as nodes as being loaded.
- *Only load 32,767*: the table contains 51,766 records. Turn this on to only load the first 32,767 records. Turn this off to see an anomaly with the Microsoft control: although you can load more than 32,767 nodes, the Nodes.Count property is a

signed 16-bit integer, so it contains -13,770. I'm not sure whether there's a problem loading that many nodes or not, such as memory corruption, but I was able to click on and expand nodes past the 32,767th. The DBI control has a similar bug: although its index pointers are 64-bit integers, during my tests, the AddNode method, which returns the index of the added node, returned a negative value once the number of nodes was over 32,000 (the actual number depended on the test).

- *Virtual mode*: this option is only applicable to the DBI control. When it's turned on, the text for each node in the TreeView is ignored. Instead, when a node is about to be drawn, the GetNode event of the control fires and expects your code to return the text to display for the specified node. This reduces the amount of memory needed by the control since the text for all those nodes isn't stored within the control. It doesn't improve performance much.

Table 1 shows the results of running these forms with various options set. All times are in seconds and with the *Only load 32,767* option turned on. "No Lock" means clicking the Start button with *Lock during load* turned off, "Lock, no WITH" means clicking the Start button with *Lock during load* turned on, "Lock, WITH" means clicking the Start (WITH) button with *Lock during load* turned on, "Virtual, Lock, WITH" means clicking the Start (WITH) button with *Lock during load* and *Virtual mode* both turned on, and "Cursor" means clicking the Start (Cursor) button with *Lock during load* turned on. You can, of course, run the forms with other combinations.

Table 1. Performance test results.

Control	No Lock	Lock, no WITH	Lock, WITH	Virtual, Lock, WITH	Cursor
Microsoft	7.74	7.83	6.32	N/A	7.47
DBI	3.51	3.38	3.23	3.11	1.34

The conclusions of these tests are:

- The DBI control is considerably faster (at least double the speed) for loading. That alone might make it worthwhile implementing.
- Locking the window speeds up loading the DBI control by less than 10%. Weirdly, it actually slowed down the Microsoft control in my tests, which is the opposite of what I've experienced before (see the last point for a possible explanation). However, the Microsoft control flickers a lot when loading if you don't do this, so it's always a good idea.
- Encapsulating the code within a WITH/ENDWITH block doesn't make as much of a difference with the DBI control as it does with the Microsoft. I think the difference is the DBI sample form uses WITH Thisform.oTree whereas the Microsoft form uses WITH Thisform.oTree.Nodes. Without the WITH statement, accessing the extra member of the Microsoft control tens of thousands of times obviously has a bigger performance impact than just accessing the control itself.

Copyright 2020, Doug Hennig

- Using virtual mode speeds up loading at the expense of a bit of complexity: needing to implement the GetNode method as discussed earlier.
- As the "Cursor" test shows, although it doesn't make much difference with the Microsoft TreeView, loading the DBI TreeView is a lot faster, double in this case, if you add nodes at the end of the TreeView rather than to existing nodes. This is likely because the TreeView has to insert rows into various arrays (NodeText, NodeData, etc.) and move existing rows down.
- I didn't show it here, but loading time is much more consistent for the DBI control than for the Microsoft control. For example, running one test five times, I got values between 7.10 and 11.30 seconds (a range of 4.2 seconds) for the Microsoft control but 3.79 to 3.99 (a range of just 0.2 seconds) for the DBI control.

Using ctxTreeView

One of the first things you'll find about ctxTreeView is that, unlike the Microsoft control, it's entirely based on indexes and (I'm guessing) arrays rather than a collection of node objects. For example, to add a node to the Microsoft TreeView, you typically use code like this:

```
loNode = Thisform.oTree.Nodes.Add(, 1, lcKey, lcNodeText, lcImage)
```

for a top-level node or:

```
loNode = Thisform.oTree.Nodes.Add(lcParentKey, 4, lcKey, lcNodeText, lcImage)
```

for a child node. loNode is an object that you can set properties on, such as:

```
loNode.ForeColor = rgb(0, 0, 0)
loNode.BackColor = rgb(255, 255, 255)
loNode.Bold      = .F.
```

Adding a node to ctxTreeView looks like this:

```
lnIndex = Thisform.oTree.AddNode(lcNodeText, 0, 1)
```

The second parameter is the position for the new node: 0 for the end of TreeView, 1 for above the selected node, or 2 for after the selected node. The third parameter is the level: 1 for a top-level node, 2 for a child node, 3 for a grandchild, etc. For example, to add a child node to an existing top-level node, first select it, then add the new node:

```
Thisform.oTree.Selected = 10   && the index of the node to add a child to
lnIndex = Thisform.oTree.AddNode(lcNodeText, 2, 2)
```

The return value is the index into the various arrays that contain properties about the nodes. For example:

```
Thisform.oTree.NodeForeColor[lnIndex] = rgb(0, 0, 0)
Thisform.oTree.NodeBackColor[lnIndex] = rgb(255, 255, 255)
Thisform.oTree.NodeFontBold[lnIndex]  = .F.
```

There are several variants to AddNode, including AddFontNode, which specifies a custom font, and AddPictureNode, which specifies an image. Like the Microsoft control, images have to be loaded first (although you don't need a separate ImageList control with ctxTreeView since it's built in) by calling AddImage. Unlike the Microsoft control, you have to reference images by index rather than key, so you have to use something like 2 instead of "Product." Here's an example:

```
Thisform.oTree.AddImage(loadpicture('Image1.bmp'))  && the first image so its index is 1
Thisform.oTree.AddPictureNode(lcNodeText, 0, 1, 1, 1, 1)
```

The last three parameters are the node image, the image when the node is selected, and the image when the node is expanded.

In addition to using an index for the arrays, many of the ctxTreeView methods expect to be passed an index. Indexes are zero-based, so the first node is node 0. The Selected property is the index of the selected node; it's -1 when no node is selected. Adding a child node automatically changes Selected to the index for the new node; adding top-level nodes doesn't do that. The reason for this is because typically you use a loop to add nodes, specifying 2 for child nodes so they're always added after the selected node, which is the previous child node added in the loop. That way, you don't have to constantly change Selected while loading child nodes.

Each node in a Microsoft TreeView has a Key property that must be unique. I usually put a record's primary key value into the Key property of its node when I fill the TreeView and then use code like this to display information about the record when the user clicks a node (SelectedItem is a reference to the node object for the selected node):

```
seek Thisform.oTree.SelectedItem.Key
Thisform.Refresh()
```

In ctxTreeView, the index of a node is variable, since it represents the position of the node in the TreeView. For example, if you add a node above node 10, the new node becomes node 10 and the former node 10 is now node 11. So, you can't rely on the node index as the node's ID. Instead, store numeric keys in the NodeData array or character keys in NodeCargo. (Note: the Microsoft control only support character keys, so you have to use TRANSFORM() on numeric values.) For example, when loading a TreeView:

```
lnIndex = Thisform.oTree.AddNode(lcNodeText, 2, 2)
Thisform.oTree.NodeData[lnIndex] = lnID
   && use NodeCargo for a character key
```

You can then do this when the user clicks a node:

```
seek Thisform.oTree.NodeData[Thisform.oTree.Selected]
   && use NodeCargo for a character key
Thisform.Refresh()
```

If you want to find which node is used for a particular record, you'd use this for the Microsoft control:

```
loNode = Thisform.oTree.Nodes[lcKey]
```

This throws an exception if lcKey isn't found.

For ctxTreeView, use the following for numeric keys:

```
lnIndex = Thisform.oTree.FindData(lnKey)
```

This returns -1 if lnKey isn't found.

Unfortunately, there isn't a FindCargo method, so for character keys, you have to use something like:

```
lnIndex = -1
for lnI = 0 to Thisform.oTree.ListCount – 1
    if Thisform.oTree.NodeCargo[lnI] == lcKey
        lnIndex = lnI
        exit
    endif
next
```

As you can guess, this won't be nearly as fast as using FindData, especially if the key you're looking for is near the end of the TreeView. A test I did with 51,785 nodes took 0.004 seconds for FindData to find the key of the last node in the TreeView while using the FOR loop with NodeCargo took 0.219 seconds, 55 times slower. So, I recommend always using a numeric ID when possible, even if it means using RECNO() for the record the node represents.

As I mentioned earlier, most developers use a "lazy loading" technique to load a TreeView control: why load 51,785 nodes if the user is only going to look at a few of them? With the Microsoft control, a "+" indicating that a node has children doesn't appear unless the node actually has children. So, the trick is to add a "dummy" child node to the parent so the "+" appears, then when the node is expanded for the first time, delete the "dummy" node and add the real children. Since this is a common technique, DBI added support for it with the NodeIsParent array. Set the value to .T. for a node to indicate that a "+" should be displayed whether there are children or not. This code is adapted from the FirstDraw event of the ctxTreeView control in TreeViewSample.scx that accompanies this document to load nodes for the records in the Categories table in the sample Northwind database (note: FirstDraw fires when the TreeView is first drawn and is sometimes used to load the TreeView):

```
scan
    lnIndex = This.AddNode(trim(CategoryName), 0, 1)
    This.NodeData[lnIndex]     = CategoryID
    This.NodeIsParent[lnIndex] = .T.
endscan
```

The following code was adapted from the Expand method of the ctxTreeView control, which fires when a node is expanded, to load the products for the selected category if they haven't already been loaded:

```
lparameters tnIndex
if This.AllChildren[tnIndex] = 0
    lnID = This.NodeData[tnIndex]
    select Products
    scan for CategoryID = lnID
        lnIndex = This.AddNode(trim(ProductName), 2, 2)
        This.NodeData[lnIndex] = ProductID
    endscan
endif
```

I ran into a peculiarity with ctxTreeView when writing this code. Initially, I checked the Children array rather than AllChildren, thinking it would be faster since AllChildren indicates all children at all levels whereas Children is just the number of direct children. When I expanded a category node, it worked as expected. However, if I collapsed the node and then expanded it again, all the child nodes disappeared. It turns out that Children returns the count of children and then deletes them all. I've reported that as a bug to DBI. In the meantime, AllChildren works just fine without this behavior.

If you have a lot of nodes, you might want to use virtual mode to save memory. To do this, set LoadType to 1. That tells the TreeView to ignore the first parameter in the AddNode method and not store the node text in the control. Instead, as a node is being drawn, the GetNode event fires, and you'll use code to set the VirtualText property to the text to display for the node. Here's an example of GetNode that looks up the record for the node and sets VirtualText to the description:

```
lparameters tnIndex, tnColumn
if This.LoadType = 1
    && need to check LoadType since this method get called even if LoadType is
    && 0 (non-virtual)
    lnID    = This.NodeData[tnIndex]
    lnLevel = This.NodeLevel[tnIndex]
    if lnLevel = 1
        select Categories
        seek lnID
        lcName = trim(CategoryName)
    else
        select Products
        seek lnID
        lcName = trim(ProductName)
    endif
    This.VirtualText = lcName
endif This.LoadType = 1
```

As the comment in the code notes, it's peculiar that GetNode is called even when you're not using virtual mode.

ctxTreeView has a lot of properties that control its appearance, some of which you can manage with its Properties dialog (right-click the control and choose "ctxTreeView Control 3.0 Properties…" from the shortcut menu; see **Figure 3**). Interestingly, you can display this dialog programmatically, even from within an EXE, by setting the PropertyPages property to 1 (it acts like a method rather than a property). I won't go into the various properties, as they're well documented.

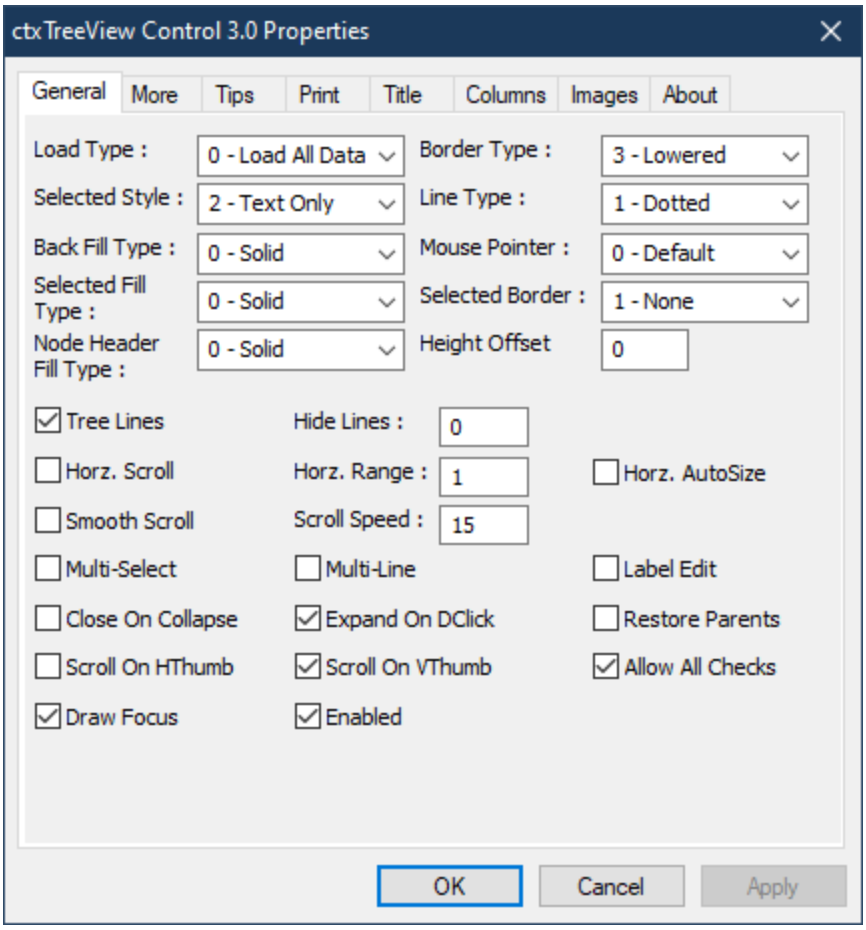

Figure 3. The ctxTreeView Properties dialog.

Some other notes about ctxTreeView:

- Setting Enabled to .F. disables the control but it doesn't visually appear to be disable. The Microsoft TreeView has the same issue.
- While ctxTreeView supports drag and drop, it's the native kind, not OLE drag and drop. That means you can't drag from outside the application (such as file from File Explorer) and drop it on a TreeView. This may be a deal-breaker for some applications.
- Typing in the TreeView doesn't do incremental searching like the Microsoft TreeView does. You can implement this yourself in code if desired.

Sample form

DBI provides several sample forms that show how the TreeView control works plus I created one you can experiment with (**Figure 4**). This form uses the Northwind sample database that comes with VFP. It displays product categories as parent nodes and the products in each category as the child nodes.

Here are some notes about this form that'll give you some idea of how ctxTreeView works.

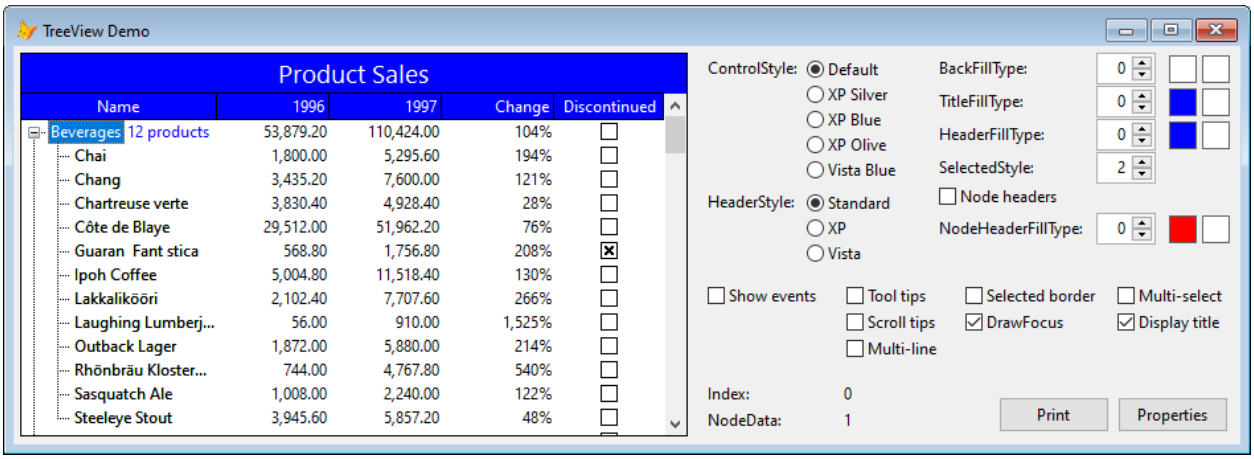

Figure 4. TreeViewSample.scx allows you to experiment with features of the ctxTreeView control.

Columns

Each node not only displays the name but also the sales for 1996 and 1997, the % change between those two years, and a checkbox indicating whether the product has been discontinued or not. The Init method of the form sets up the multiple columns in the TreeView using this code:

```
with This.oTree
    lnIndex = .AddColumn('Name', 160)
    .ColumnSortable[lnIndex] = .T.

    lnIndex = .AddColumn('1996', 85)
    .ColumnTextAlign[lnIndex] = AlignRight
    .ColumnSortable[lnIndex] = .T.

    lnIndex = .AddColumn('1997', 85)
    .ColumnTextAlign[lnIndex] = AlignRight
    .ColumnSortable[lnIndex] = .T.

    lnIndex = .AddColumn('Change', 85)
    .ColumnTextAlign[lnIndex] = AlignRight
    .ColumnSortable[lnIndex] = .T.

    lnIndex = .AddColumn('Discontinued', 85)
    .ColumnCheckBox[lnIndex] = 1
endwith
```

To load values into each column, set the CellText property of the row and column. This code was taken from FirstDraw when category nodes are added:

```
lnIndex = This.AddNode(trim(CategoryName), 0, 1)
This.CellText[lnIndex, 2] = transform(N_1996, '999,999.99')
This.CellText[lnIndex, 3] = transform(N_1997, '999,999.99')
lnChange = iif(N_1996 = 0, 100, (N_1997 - N_1996)/N_1996 * 100)
This.CellText[lnIndex, 4] = transform(lnChange, '9,999%')
```

Setting ColumnSortable to .T. makes a column sortable. Click any of the first four columns to sort on that column in ascending order and click again to toggle to descending order. Here are some notes about sorting:

- Child nodes are sorted within their parent rather than within the entire TreeView.
- Any child nodes you add after sorting aren't sorted, so this form uses the following code after loading child nodes in the Expand event to resort the TreeView:

```
if This.SortColumn > 0
   This.Sort(This.SortColumn, .T.)
endif This.SortColumn > 0
```

- Sorting changes the NodeIsParent status of any nodes that don't really have any children to .F., which means the "+" for an parent node that hasn't been expanded yet disappears. To prevent that, set the RestoreParents property to .T.

Since the value of the checkbox comes from a field in the Products table, turning on or off the checkbox should update the record in the table. Code in the CheckClick event shows how to do that. (Don't worry—it's commented out in the sample form so it doesn't change your VFP sample data.)

Resizing the TreeView enlarges or shrinks the last column, which isn't likely what you want (usually it's the first column you want adjusted), so this code in the Resize event forces the last four columns to a specific width and sizes the first one to the remaining width:

```
This.ColumnWidth(2) = 85
This.ColumnWidth(3) = 85
This.ColumnWidth(4) = 85
This.ColumnWidth(5) = 85
This.ColumnWidth(1) = This.Width - 85 * 4 - sysmetric(5)
   && account for the scroll bar
```

ToolTips

Turn on *ToolTips*, which sets the TipsDisplay property to .T., to display tooltips. You can display line tips, which are tooltips for nodes, parent tips, which display tooltips about the parent node for the current child when the parent node has scrolled off screen and the mouse is over the root line for the child, or both by setting the TipsType property. Specify the text to display for the tooltip by setting the TipsText property in the SetTips event. In the demo form, the line tip for a category shows its description and for a product shows some details about the product, including the quantity in stock.

Turn on *Scroll tips*, which sets the TipsOnScroll property to .T. and the ScrollOnVThumb property to .F., to display a tooltip for the current node as you drag the vertical scroll thumb down. It's sort of weird behavior because the TreeView doesn't appear to scroll when you do that, redrawing itself to show the current set of nodes once you let go of the mouse button.

Note that scroll tips display multiple lines if you use a carriage return (CR) or line feed (LF). Line tips stop at the first CR or LF while parent tips ignore them.

Node headers

Turn on *Node headers* to display category nodes as node headers. Node headers don't display column formatting (all but the first column are blank) and have their own color and font settings, so they provide a nice way to make certain nodes stand out. To make a node into a node header, set NodeHeader[Index] to .T.

One oddity in this form that you likely wouldn't see in other forms that don't switch nodes between being node headers and regular nodes is that the content of all of the columns gets squished into the first column, so the code in the Click event of the checkbox takes care of that.

Events

Turn on the *Show events* checkbox to echo event messages to the screen so you can see when they fire.

Contrary to its name, the NodeClick event fires even when you click part of the TreeView that isn't a node, such as the title. Use the Change event if you only want events when the user clicks a different node than the one already selected.

Other notes

- Turn on *Multi-line* to display each node over multiple lines. This is handy if the nodes contain a lot of text.
- Turn on *Multi-select* to allow you to select multiple nodes using Shift-Click.
- Click the Properties button to programmatically display the Properties dialog. Make some changes and click Apply to see them take effect.
- Click the Print button to print the current contents of the TreeView.
- Play with the different style, fill type, and color controls to see how they affect the appearance of the TreeView.

ctxListBar

At Southwest Fox 2010, I presented a session showing Emerson Santon Reed's ThemedControls VFPX project (https://github.com/VFPX/ThemedControls). DBI's ctxListBar provides a control with a similar appearance to Emerson's ThemedToolBox and ThemedOutlookNavBar controls but a lot more powerful and a lot more customizable.

As you can see in **Figure 5**, the ctxListBar control allows you to present an Outlook-like interface (although as discussed later, it can also appear like a Toolbox-like control). ListBars consist of "lists", which are what DBI calls the different panels ("Customers," "Products," etc.) shown in Figure 5. A list can contain text, images, or other controls, such as the ctxTreeView control shown in the Customers list.

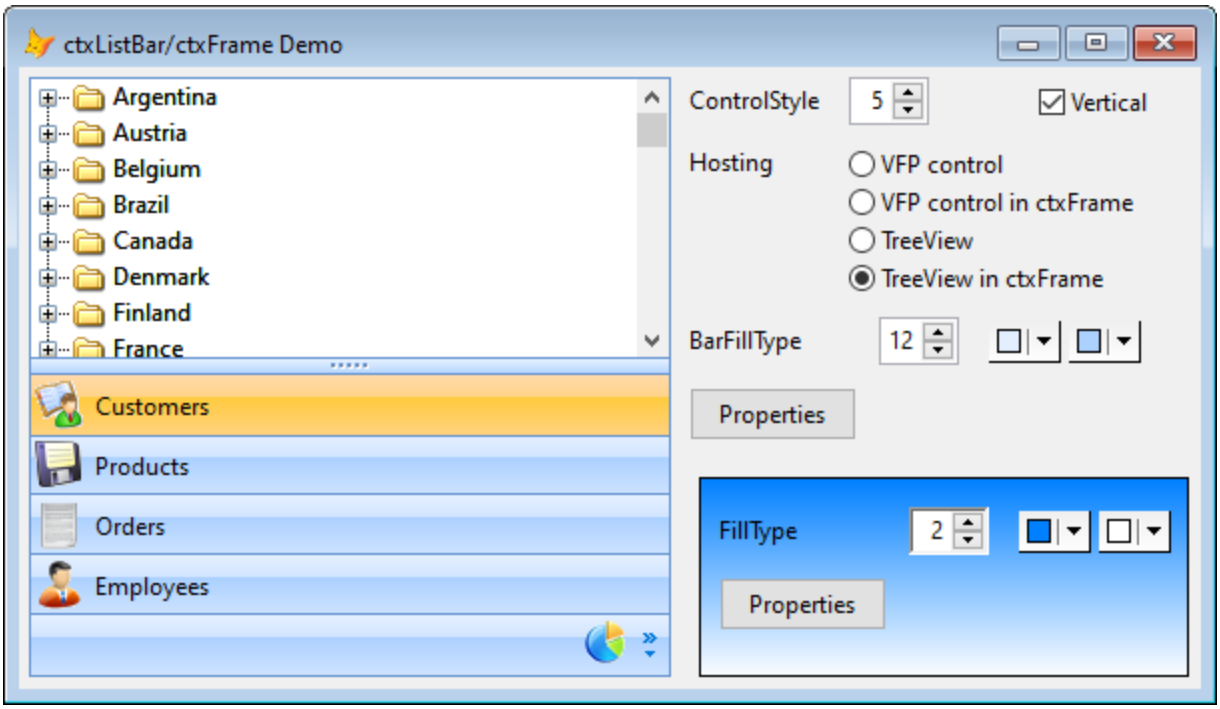

Figure 5. The ctxListBar control allows you to present an Outlook-like interface.

Here are some of the features of ctxListBar:

- There are five different control styles, set by the ControlStyle property, as shown in **Figure 6**. As you can see, values 1 and 2 look similar to the navigation control in the VFP Toolbox while values 3 through 5 resemble an Outlook navigation bar. In addition to a little different appearance between 1 and 2, another difference is that standard doesn't display text items in lists, only images.

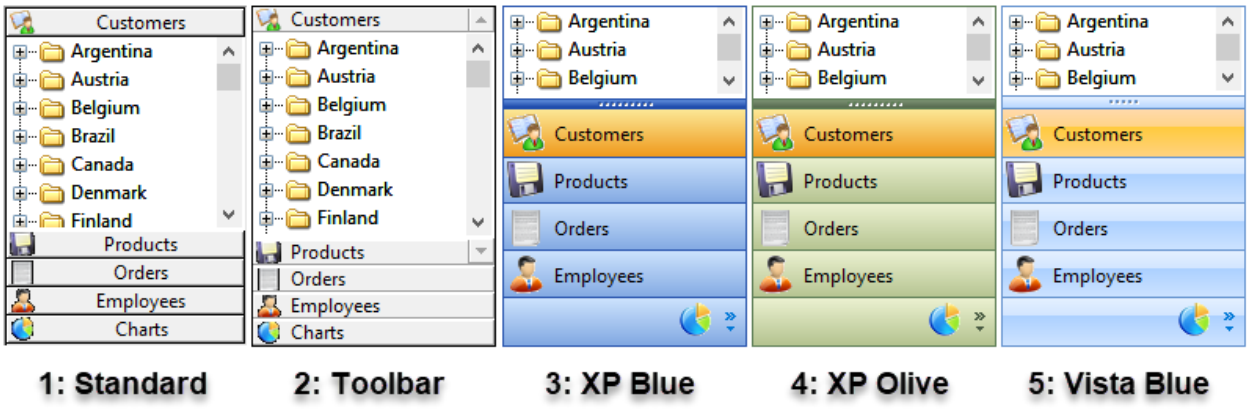

Figure 6. The ControlStyle property controls the appearance.

- Setting ControlStyle sets the default appearance for that style type but you can further customize the appearance by setting various properties. For example, although ControlStyle = 5 gives a Vista Blue appearance, you can set BarFillType,

BarForeColor, BarBackColor, and BarBackColorTo to customize the appearance (**Figure 7**).

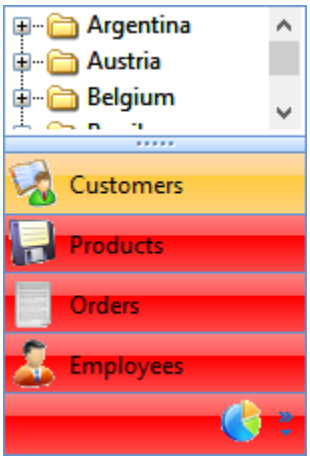

Figure 7. You can change the default colors for a particular ControlStyle setting to customize the appearance.

- The ListBarStyle property can be 0-Horizontal (**Figure 8**) or the default 1-Vertical.

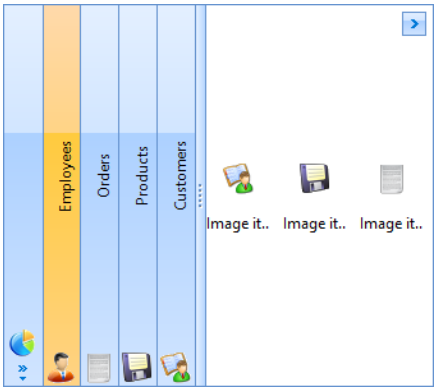

Figure 8. A horizontal ctxListBar.

- Items can be displayed vertically or in columns.
- Like ctxTreeView, ctxListBar has a built-in image list which you either load through its Properties dialog at design time or programmatically using AddImage at run time.
- It supports ToolTips for items.
- You can control how a selected item appears, including when the mouse pointer is over it.
- You can programmatically display the ListBar Properties dialog, even within an EXE.
- You can persist settings and items to XML.

Using ctxListBar

As with ctxTreeView, ctxListBar uses arrays for lists, items, and images. For example, to load an image, add a Products list, and set its image, use code like this:

```
Thisform.oListBar.AddImage(loadpicture('Products.bmp')
lnIndex = Thisform.oListBar.AddList('Products')
Thisform.oListBar.BarClip[lnIndex] = 1
```

Strangely, while ctxTreeView indexes are 0-based, in ctxListBar they're 1-based.

The ListBar has one list automatically. To set its text, you could set oListBar.BarText[1] but you can also set the Caption property.

Images for items are limited to 32x32. The size to use for list images (the ones displayed in the bar) depends on the BarHeight property, which defaults to different values depending on the setting of ControlStyle but can be changed as necessary. The sample form accompanying this document uses 16x16 images when ControlStyle is 1 or 2 and 24x24 for the other values. Oddly, images can only be BMP, CUR, ICO, DIB, EMF, or WMF formats; PNG, JPG, and GIF aren't supported. You'll likely want to set MaskBitMap to .T. and MaskColor to the background color of the images you use so the image background doesn't appear.

To add an item to a list, call either AddItem for a text-only item, AddImageItem for an item using an image in the built-in image list, or AddPictureItem to specify an external image. Pass these methods the index of the list to add the item to, the text of the item, and for AddImageItem or AddPictureItem, either the index of the image or an image object returned by the VFP LOADPICTURE() function. Here's an example:

```
with This.oListBar
    lnIndex = .AddList('Employees')
    .BarClip[lnIndex] = 9
    .AddImageItem(lnIndex, 'Image item 1', 1)
    .AddImageItem(lnIndex, 'Image item 2', 2)
    .AddImageItem(lnIndex, 'Image item 3', 3)
    .AddImageItem(lnIndex, 'Image item 4', 4)
    .AddImageItem(lnIndex, 'Image item 5', 5)
endwith
```

Text items are for display only; nothing happens when you click them. Clicking an image item fires the ItemClick event.

Hosting controls

ctxListBar can sort of host other controls besides text and image items. I say "sort of" because the controls aren't really hosted inside the ctxListBar but are drawn as if they are. One complication in VFP is that because an ActiveX control is its own window, you can't put a VFP control on top of an ActiveX control. The consequence of this is that you can't directly host VFP controls inside a ctxListBar. Instead, you have to use a ctxFrame control as a container for the VFP controls, then host the ctxFrame object in the ctxListBar.

You can see this behavior in the sample form accompanying this document. When you choose "VFP control" for the "Hosting" setting, an option group is "hosted" in the Customer list, and yet nothing appears. That's because the option group is behind the ListBar. When you choose "VFP control in ctxFrame," an option group appears in the Customer list because it's contained in a ctxFrame control, which can appear above the ListBar. Interestingly, even another ActiveX control, such as ctxListView, exhibits this same behavior; choosing "TreeView" doesn't display anything but choosing "TreeView in ctxFrame" does.

The "hosting" (really, drawing) is handled in two events of the ctxListBar control:

- ListResize, which fires when the list area of the ctxListBar changes. Set the Top, Left, Height, and Width properties of the ctxFrame control so it's positioned over top the ctxListBar.
- ListChange, which fires when a new list becomes active. Set the Visible property of the ctxFrame control accordingly.

In the sample form, ListResize sets the position and size of four controls (matching the four values of the Hosting control) and ListChange sets the Visible property of these controls so the proper one appears depending on which list is selected and the setting of "Hosting."

Sample form

DBI provides several sample forms that show how the ListBar control works plus I created one you can experiment with, ListBarDemo.scx (Figure 5). This form uses the Northwind sample database that comes with VFP. The Customers list displays the hosted control specified in the Hosting setting; if you choose "TreeView in ctxFrame," it displays a TreeView showing countries, customers in those countries, orders for each customer, and the products in each order. The Products list displays a TreeView with product categories as parent nodes and the products in each category as the child nodes. The Orders list displays some sample text items and the Employees list displays some sample image items. The Chart list doesn't contain anything.

Try changing the various settings in the form to see how the ListBar appears depending on your choices. Notice that you can change the BarFillType setting and the colors used for the bar to customize a particular control style. You can also click the Properties button to programmatically display the ctxListBar Properties dialog to see how various settings work interactively.

ctxFrame

ctxFrame is similar to Emerson's ThemedContainer: it's a container that can contain other controls but also provides special effects such as gradients without having to use something like GDIPlusX for drawing. The ListBarDemo.scx sample form discussed in the previous section also demonstrates the use of ctxFrame.

Using ctxFrame

Add controls to a ctxFrame object as you would a VFP container: right-click the object, then drag or paste controls into it. ctxFrame has three borders: the border around the control, an outer border (which is inside the border), and an inner border. You can control the appearance of the borders with the following properties: BorderColor and BorderStyle, which affect the surrounding border; OuterBorder, OuterWidth, InnerBorder, InnerWidth, and DistanceApart, which determine what the outer and inner borders look like and how far apart they are; and ShowBottom, ShowLeft, ShowRight, and ShowTop, which control which specific borders appear. FillType determines whether a solid fill or one of various gradient fills is used, and BackColor and BackColorTo determine the fill colors. You can display drag bars, which allow the frame to mimic a toolbar, with the DragBars and DragBarColor properties.

ctxDropMenu

I haven't used VFP's native menus directly in many years. I always hated the fact that the Menu Designer is a clunky tool and that menus are hard-coded procedural code rather than object-oriented. Because I wanted a more flexible menuing system, I created a set of OOP menu classes that are now part of VFPX (https://github.com/VFPXHome/OOPMenu). Internally, they're just wrappers for the VFP DEFINE MENU, DEFINE PAD, DEFINE BAR, and other menu-related commands, but at least I can manipulate my menus as objects now.

However, the other issue is that, as you can see in **Figure 9**, VFP's native menus (the one on the left) look out of date compared to menus in more recent applications, such as Microsoft Outlook (the one on the right).

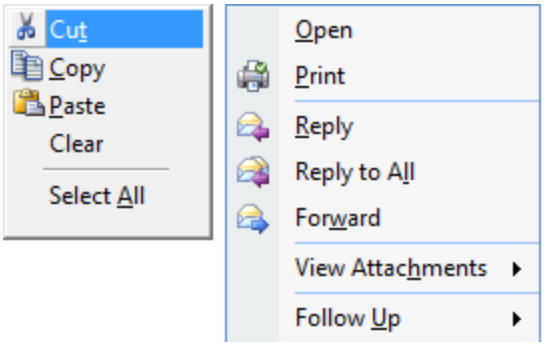

Figure 9. VFP's native menus look out of date compared to newer applications.

The VFPX PopMenu project (https://github.com/VFPX/PopMenu) provides an entirely new way of doing menus. Not only are they object-oriented, they also use the native Windows menuing system rather than VFP's menuing system, meaning that your menus can look just like those in Microsoft Office applications. One downside of PopMenu, though, is that there's no documentation and all comments (in code and the Properties window) are in Chinese.

DBI provides an ActiveX control named ctxDropMenu that provides similar features to PopMenu but has actual documentation. As you can see in **Figure 10**, ctxDropMenu allows you to create shortcut menus that look those in modern applications.

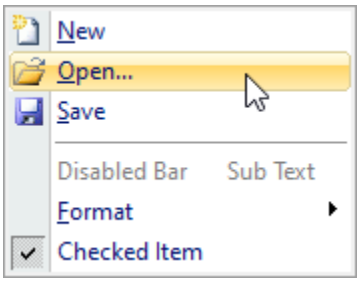

Figure 10. ctxDropMenu allows you to create shortcut menus that look like those in modern applications.

You can customize just about everything about the appearance of the menu, including the colors of:

- the margin (the strip at the left of the menu)
- the menu as a whole
- individual bars
- the selected bar

Using ctxDropMenu

As with all DBI controls, ctxDropMenu has its own image list and uses arrays for the properties of items. For example, to load an image, add an Open bar (not that kind of open bar ☺), and set its image, use code like this:

```
Thisform.oMenu.AddImage(loadpicture('Open.bmp'))
lnIndex = Thisform.oMenu.AddItem('&Open...', 0, 1)
Thisform.oMenu.ItemClip[lnIndex] = 1
```

Pass AddItem the caption for the bar ("&" indicates the hot key for the bar similar to how "\<" does in VFP menus), 0 for a regular bar or 1 for a separator bar, and the level (1 for a bar in the menu, 2 for a bar in a submenu, and so on). The ItemClip array contains the image number for each bar.

As with ctxListBar, images can only be BMP, CUR, ICO, DIB, EMF, or WMF formats. You'll likely want to set MaskBitMap to .T. and MaskColor to the background color of the images you use so the image background doesn't appear.

There are lots of properties that control the appearance of the menu. The Margin* properties (such as MarginBackColor and MarginFillType) affect the margin, the Select* properties (such as SelectBackColor and SelectFillType) affect a menu item when the mouse is over it, and the Menu* properties (such as MenuBackColor and MenuFillType) affect the menu as a whole. The Item* properties (such as ItemBackColor and ItemFillType) are arrays that affect the item specified by the index.

To display the menu, call either DropMenuAtCursor to show the menu at the mouse pointer position or DropMenu(x, y) to show the menu at the specified position. The ItemClick event fires when an item is selected, with the item number passed as a parameter, so take the appropriate action, likely in a CASE statement.

The code shown in **Listing 1** creates the menu shown in Figure 10.

Listing 1. Code to create the menu shown in Figure 10.

```
with This.oFormMenu

* Load the images.

    .AddImage(loadpicture('New.bmp'))            && 1
    .AddImage(loadpicture('Open.bmp'))           && 2
    .AddImage(loadpicture('Save.bmp'))           && 3
    .AddImage(loadpicture('AlignHeightMin.bmp')) && 4
    .AddImage(loadpicture('AlignWidthMin.bmp'))  && 5

* Give the menu an Office-like appearance.

    .MaskBitmap = .T.
    .MaskColor  = rgb(255, 255, 255)

    .MenuBackColor     = rgb(255, 255, 255)
    .MenuForeColor     = rgb(  0,  21, 110)

    .MarginBackColor   = rgb(233, 238, 238)
    .MarginFillType    = 0
    .MarginLineColor   = rgb(197, 197, 197)

    .SelectBackColor   = rgb(255, 253, 233)
    .SelectBackColorTo = rgb(255, 214, 105)
    .SelectBorderColor = rgb(219, 206, 153)
    .SelectFillType    = 12
    .SelectForeColor   = rgb(  0,  21, 110)
    .SelectBorderType  = 6
    .LimitSelect       = .F.

* Add the menu items.

    lnIndex = .AddItem('&New', 0, 1)
    .ItemClip[lnIndex] = 1

    lnIndex = .AddItem('&Open...', 0, 1)
    .ItemClip[lnIndex] = 2

    lnIndex = .AddItem('&Save', 0, 1)
    .ItemClip[lnIndex] = 3

    .AddItem('', 1, 1)
       && separator bar

    lnIndex = .AddItem('Disabled Bar', 0, 1)
    .ItemSubText[lnIndex] = 'Sub Text'
    .ItemEnabled[lnIndex] = .F.
```

```
    .AddItem('&Format', 0, 1)
        .AddItem('Adjust to &Highest', 0, 2)
        lnIndex = .AddItem('Adjust to &Shortest', 0, 2)
        .ItemClip[lnIndex] = 4
        .AddItem('Adjust to &Widest', 0, 2)
        lnIndex = .AddItem('Adjust to &Narrowest', 0, 2)
        .ItemClip[lnIndex] = 5

    lnIndex = .AddItem('Checked Item', 0, 1)
    .ItemChecked[lnIndex] = .T.
endwith
```

Sample form

DBI provides a single sample form that shows how ctxDropMenu works so I created one you can experiment with, DropMenu.scx (**Figure 11**). Right-click the textbox to see its shortcut menu. Try changing the various settings in the form to see how the menu appears depending on your choices. You can also click the Properties button to programmatically display the ctxDropMenu Properties dialog to see how various settings work interactively. Right-click the form to see the menu shown in Figure 10.

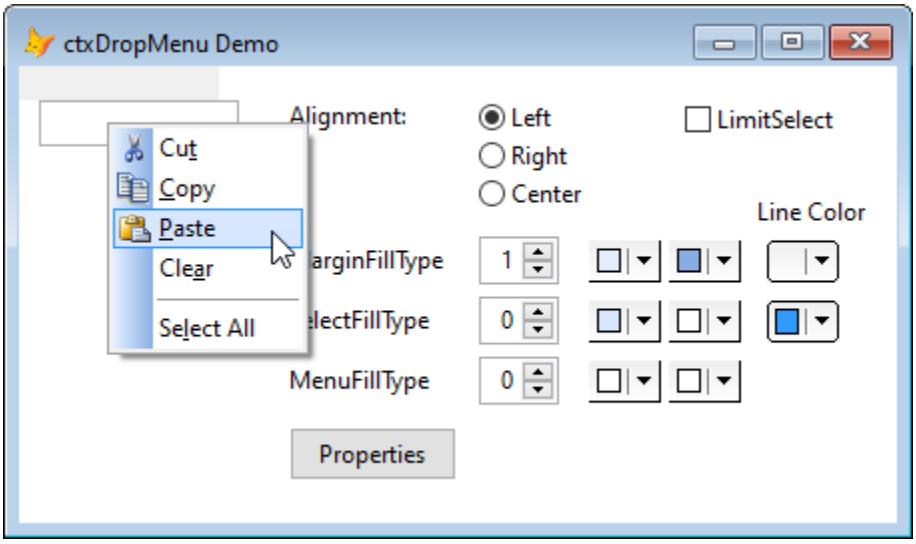

Figure 11. DropMenu,scx demonstrates how to use ctxDropMenu.

One interesting thing I found when I created this form is that there's a 100 pixel wide, 16 pixel high grey shape at 0, 0 in the form; you can see this if you change the BackColor of the form to white. I assume that's an artifact of ctxDropMenu, so don't put anything in that area or it'll be overlapped by that shape.

ctxToolBar

ctxDropMenu provides shortcut menus, but what about menu bars? Although its name suggests it's for providing toolbars, ctxToolBar actually provides menu bars as well. As far as ctxToolBar is concerned, the only difference between a toolbar and a menu bar is that the latter has subitems (the items that appear when you click a menu "pad") and no images

for the buttons (although as we'll see, you can create a dropdown button in a toolbar that has subitems). **Figure 12** shows a form with two ctxToolBar controls, the top one providing a menu and the one under it providing a toolbar.

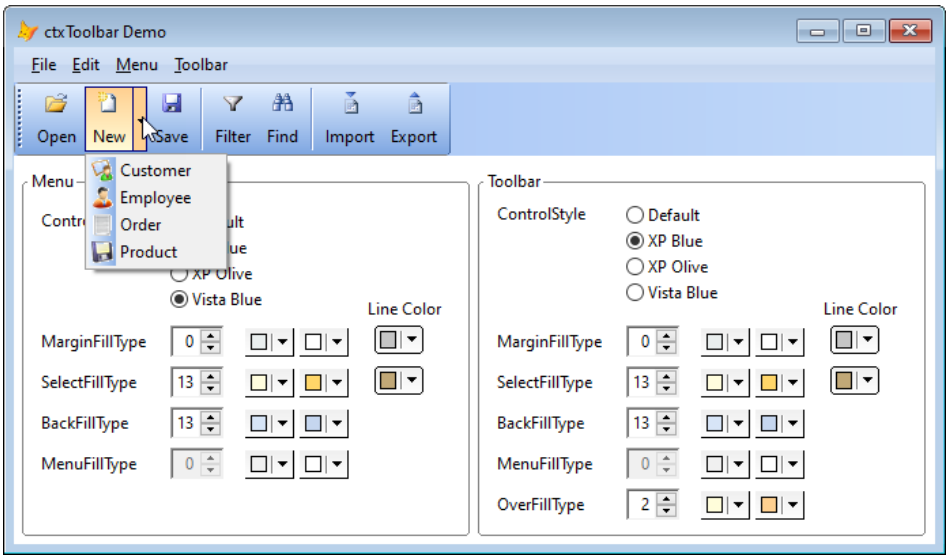

Figure 12. ctxToolBar provides both menus and toolbars.

One cool feature of ctxToolBar is that it has a built-in menu designer; the Menu Builder page of its Properties dialog (**Figure 13**) allows you to specify items and their properties visually. So, you can either create your menu in code at run time or visually at design time.

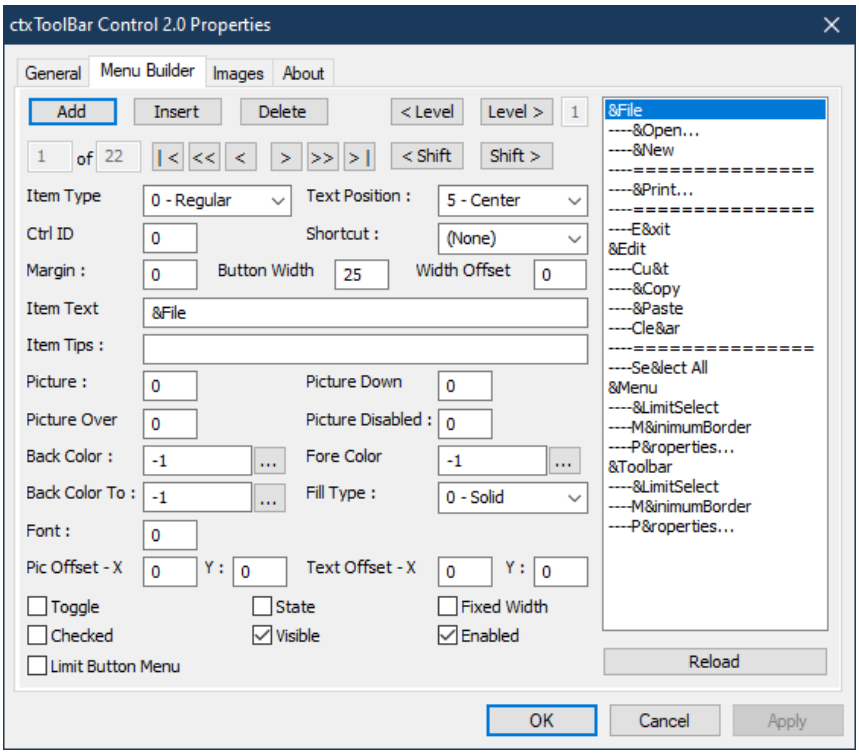

Figure 13. The Menu Builder page of the ctxToolBar Properties dialog provides a visual menu designer.

Using ctxToolBar

Working with ctxToolBar is very similar to working with ctxDropMenu: add images using AddImage, add items using AddItem, and set item properties by assigning values to various array elements. Control the menu or toolbar appearance with the Margin*, Select*, Menu*, and Item* properties. There are quite a few more properties with ctxToolBar, however. Ones you'll likely work with are:

- ItemTextPosition: set this to the desired position (0 = none, 1 = bottom, 2 = top, 3 = left, 4 = right, and 5 = center) for the item text. For a menu bar, it'll typically be 5 and for a toolbar it's often 1 so the text appears below the image.
- ControlStyle: as we've seen with ctxTreeView and ctxListBar, this applies a "theme" to the control, automatically setting numerous properties at once to give the desired appearance. You can see how both the menu and toolbar have ControlStyle set in Figure 12.
- OverForeColor, OverFillType, OverBackColor, and OverBackColorTo: these properties control the appearance of an item when the mouse is over it. This can give a nice visual effect as the mouse pointer is moved over toolbar buttons.
- DragBars: determine whether a "drag bar" appears as the left edge of a toolbar (it likely doesn't make sense to use it with a menu bar), and if so whether it's a single line, a double line, or a set of dots, as it is in Figure 12.
- MinimumBorder: notice in Figure 12 that the menu bar extends all the way across the form but the toolbar has a border after the Export button even though the blue band of the toolbar extends across the form. The difference is that the menu bar has BorderType set to 1-None while the toolbar has it set to 0-Regular and MinimumBorder set to .T. Setting MinimumBorder to .F. still gives the toolbar a border but it isn't drawn after the last button.
- TipsDisplay and ItemTips: to display tooltips on toolbar buttons, set TipsDisplay to .T. and the ItemTips array element for each button to the desired text. You can control how the tooltips appear with the various Tips* properties, such as TipsFont and TipsBackColor.
- ItemLevel: to create a dropdown button in a toolbar like the New button in the sample form, specify 1 as the first parameter for AddItem. Then use AddItem to add regular or separator items to the toolbar but set ItemLevel for those items to 2 so they appear as subitems.

A couple of other things about ctxToolBar:

- It doesn't act like exactly a VFP toolbar: clicking a button in the toolbar causes the current control to lose focus.
- The toolbar supports drag and drop so you can add and remove items to it at run time. This might be useful if you want to give your users the ability to customize

their toolbars. The ctxToolBarDrag sample that comes with DBI's controls shows an example of this.

Sample form

DBI provides numerous sample forms that show how ctxToolBar works. I created one you can experiment with, ToolbarDemo.scx. Try changing the various settings in the form to see how the menu and toolbar appear depending on your choices. You can also change the LimitSelect and MinimumBorder properties in the Menu and Toolbar menus or display the ctxToolBar Properties dialog by choosing the Properties menu item.

ctxDropDate

ctxDropDate provides a date picker control that's vastly superior to the Microsoft Date and Time Picker Control. ctxDropDate has the following features:

- It has a more modern appearance than the Microsoft control.
- It has both drop-down calendar and optional spinner buttons that increment or decrement the date.
- You can configure just about everything about the calendar: font, colors, month and day names, whether week numbers appear, and so on.
- It support various numerical date formats (mm/dd/yyyy, dd/mm/yyyy, yyyy/mm/dd) as well as text format (such as "February 20, 2020").
- It supports intelligent data entry. When a numerical format is used, you can enter + or – and a value to increment or decrement the date by that many days. With a text format, you can enter something like "j 21" and have it expand to "January 21, 2020.)

The calendar control acts more like the calendar you see in other modern applications than the Microsoft control does: clicking the title displays a month picker, clicking it again displays a year picker, and clicking it again displays a decade picker.

One thing the Microsoft control can do that the DBI control can't do is edit the time portion of a DateTime value.

Using ctxDropDate

One weird thing about ctxDropDate is that it doesn't have a property containing the selected date. Instead, Value (and Date, which contains the same value as Value) contains the number of days since the base date. The base date for the control is January 1, 1900 but the base date for VFP is December 30, 1899. The difference between those dates is stored in the DateOffset property. So, to convert Value into a VFP date, use:

```
ldVFPDate = oControl.Value + date(1900, 1, 1) – oControl.DateOffset
```

To convert a VFP date into Value, use:

```
This.Value = ldVFPDate - date(1900, 1, 1) + This.DateOffset
```

To make things more complicated, the Value property isn't available until after the control is first displayed so if you want to set an initial date value, set the Date property instead.

ctxDropDate doesn't support data binding (it doesn't have a ControlSource property) but you can simulate that by setting Value to the value of the control source in the Refresh event and assigning Value to the control source in the Change event.

Some of the properties you'll likely want to set are:

- ControlStyle: as we've seen with other DBI controls, this applies a "theme" to the control, automatically setting numerous properties at once, such as DisplayStyle, to give the desired appearance.
- ButtonStyle: set it to 0 for "regular" style drop and spin buttons or 1 for "XP" style buttons.
- DisplayStyle: set it to 0 to always display a border around the control ("regular" style) or 1 to only display the border when the control has focus or the mouse pointer is over it ("flat" style).
- Font properties: Font controls the font for the control while CalFont, CalTitleFont, and CalWeekFont controls the font for the calendar body, calendar title, and week numbers.
- FormatType: 0 means mm/dd/yyyy, 1 means dd/mm/yyyy, and 2 means yyyy/mm/dd.
- LongYear: set this to .F. (the default is .T.) to use 2-digit years.
- MonthFormat: use 0 for a numerical date format, 1 for a text format with the full name of the month displayed, or 2 for the first three characters of the month name.
- SpinButton: set this to .T. to display the spinner buttons. Related members are IncrementValue, which contains the number of days to increment or decrement the date by; InitialDelay, which is how many milliseconds to wait before repeatedly changing the value when a spin button is held down; RepeatRate, which is the time in milliseconds between changes when a spin button is held down; and ClickSpin, which is the event that fires when a spin button is clicked or held down.
- MaxValue and MinValue: these properties allow you to constrain the date picker to a specific date range. Related to these is NegativeDates: set that to .T. to allow dates before the base date from being entered.

Sample form

In addition to the single sample form DBI provides for ctxDropDate, I've created one called DropDate.scx (**Figure 14**) which shows both Microsoft and DBI controls. Click the Properties button to display the ctxDropDate Properties dialog and try changing settings such as turning SpinButton on or setting MonthFormat to 1-Long.

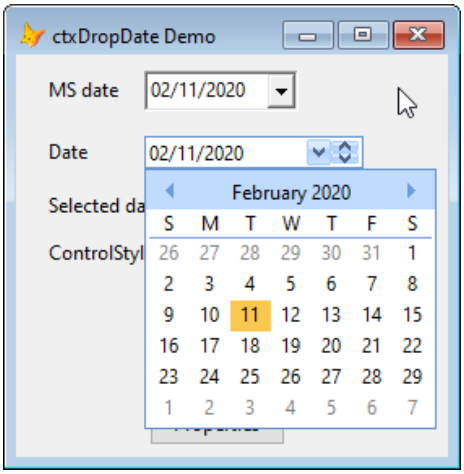

Figure 14. ctxDropDate provides a more modern date picker than the Microsoft control.

ctxCombo

ctxCombo is like a VFP ComboBox on steroids. It supports the following features:

- Customizable: you can change the fonts, styles, and colors for just about every element of the combobox. This allows you, for example, to make a combobox that looks like one in a modern application.
- Headers: you can have headers within the dropdown list to visually group items. For example, in **Figure 15**, cities are grouped by country, with the country as the header. Headers can't be selected.

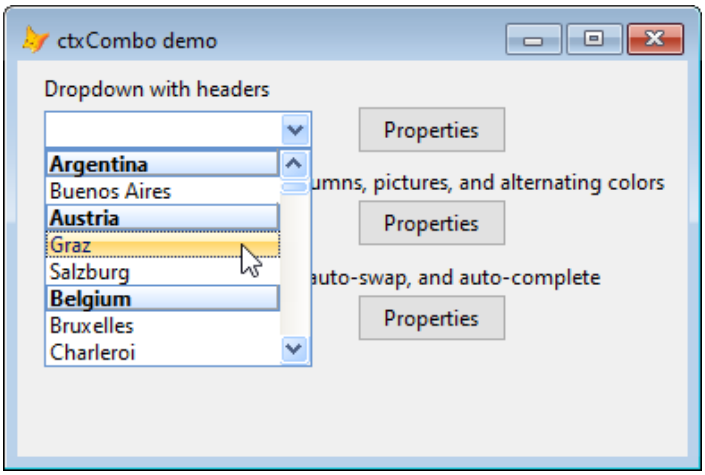

Figure 15. ctxCombo supports headers within the list.

- Columns: like the VFP ComboBox, ctxCombo supports multiple columns, but as you can see in **Figure 16**, columns can have non-scrolling headers (which have their own border, font, colors, and so on) and support other customizations such as text alignment in each column.

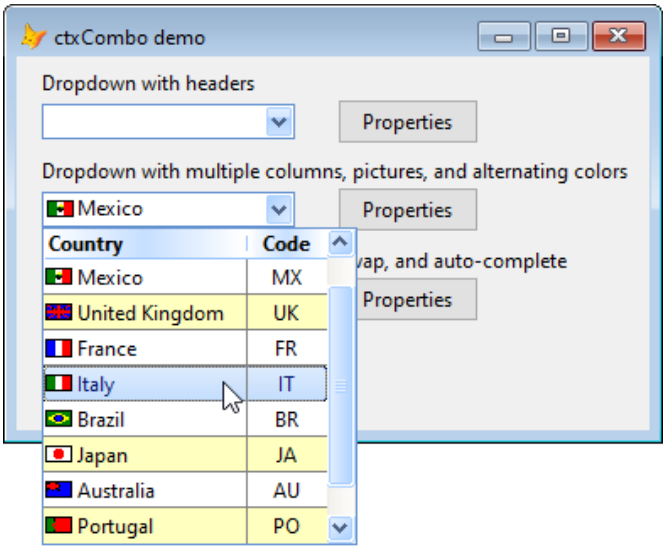

Figure 16. ctxCombo's columns can have headers and are much more customizable.

- Alternating colors: the ctxCombo in Figure 16 has alternating colors for every item. You can control the colors and how many items are in each set.
- Auto-completion: like the VFP TextBox, ctxCombo supports auto-completion. You don't get your own list of values in a table like you do in VFP (although you could support that in code) but you do get one of four modes: none, append (the first matching item appears in the text part of the combobox), suggest (all matching items appear in the list part of the combobox), or both. You can control how many characters the user has to enter before auto-completion kicks in by setting the AutoFindSize property.
- Auto-save: if a value the user types into a VFP ComboBox isn't in the list of values and you want it added to the list, you have to write code to support that. With ctxCombo, you just set the AutoSave property to .T. You do have to write code to persist any new values, though.
- Most recently used (MRU): setting the AutoSwap property to .T. causes the selected item to automatically move to the top of the list. Again, you have to write code if you want to persist the order of the items in the list.

Using ctxCombo

ctxCombo doesn't support data binding but you can simulate that by setting ListIndex to the index of the item matching the control source in the Refresh event and assigning some value (such as Text or the value of the desired column of the selected item) to the control source in the Change event.

Adding images and items works the way it does with other DBI controls we've looked at: using AddImage and AddItem. To create multiple columns, pass AddColumn the caption for the column and the column width in pixels. When you have multiple columns, separate the

text for each column with line feeds (CHR(10)), or whatever character you wish by setting the BreakChar property to the desired character. For example:

```
Thisform.Combo.AddColumn('Country', 120)
Thisform.Combo.AddColumn('Code', 50)
Thisform.Combo.AddItem('Canada' + chr(10) + 'CA')
```

To assign an image to an item, set the ComboItemPicture array element for the item to the index of the image:

```
Thisform.Combo.AddImage(loadpicture('CanadaFlag.bmp')
Thisform.Combo.AddImage(loadpicture('USFlag.bmp')
lnIndex = Thisform.Combo.AddItem('Canada')
Thisform.Combo.ComboItemPicture[lnIndex] = 1
lnIndex = Thisform.Combo.AddItem('United States')
Thisform.Combo.ComboItemPicture[lnIndex] = 2
```

If each item has a different image and you want the textbox to display the image associated with the selected item, use code like this in the Change event:

```
lparameters tnListIndex
This.Picture = This.ListImage[tnListIndex + 1]
```

Some of the properties you'll likely want to set are:

- ControlStyle: as we've seen with other DBI controls, this applies a "theme" to the control, automatically setting numerous properties at once to give the desired appearance.
- ButtonStyle: set it to 0 for a "regular" style drop button or 1 for an "XP" style button.
- DisplayStyle: set it to 0 to always display a border around the control ("regular" style) or 1 to only display the border when the control has focus or the mouse pointer is over it ("flat" style).
- Font properties: Font controls the font for the control while HeaderFont controls the font for headers in the list.
- Style: like VFP, you can make the combobox into a dropdown list or a dropdown combo with this property.
- AlternateColor: set this to .T. to alternate colors between the colors in the AltColorEven and AltColorOdd properties. AlternateItems, which defaults to 1, controls the number of items in a set.
- AutoComplete: set it to 0-None, 1-Append, 2-Suggest, or 3-Both
- AutoSave: set it to .T. to support auto-adding new items to the list.
- AutoSwap: set it to .T. to support the MRU feature.
- Header* and Select* properties: these properties allow you to specify the colors, border, and fill type for the selected item and for header items.

Sample form

DBI ships three sample forms that show how to use ctxCombo and this document comes with one I've created called ComboDemo.scx. It shows three comboboxes. The first displays how to set up list item headers (Figure 15), using an example of cities grouped by country with the country as the header. The second shows how to implement multiple columns, images on individual items plus in the text portion of the combobox, and alternating colors (Figure 16). The third shows how auto-save, auto-swap, and auto-complete works. Choose a name from the combobox and notice that it moves to the top of the list. Type a new name such as "Rick Borup" and notice that it's automatically added to the list. Type "Rick <space>" and notice how auto-completion displays the two matching names (Rick Schummer and Rick Borup) and fills in the text portion with the first match.

ctxTips

Carlos Alloatti's ctl32 library has a lot of controls you can add to your VFP applications to make them look more modern (see my "Using ctl32 to Create a More Modern UI" white paper at http://doughennig.com/papers.aspx). Unfortunately, Carlos' web site is long gone so there's little documentation and these controls are no longer maintained.

One of the controls I use from his collection is ctl32_BalloonTip, which acts like a tooltip in that it appears automatically when you hover the mouse pointer over a control but displays a lot more information and is more attractive than a tooltip. For example, in **Figure 17**, the balloon tip displays the properties of the field under the mouse that've been changed from their defaults, saving me from having to click the Properties button just to see how the field is set up. Other uses are to display pop-up help information for a control, display status information, and so on.

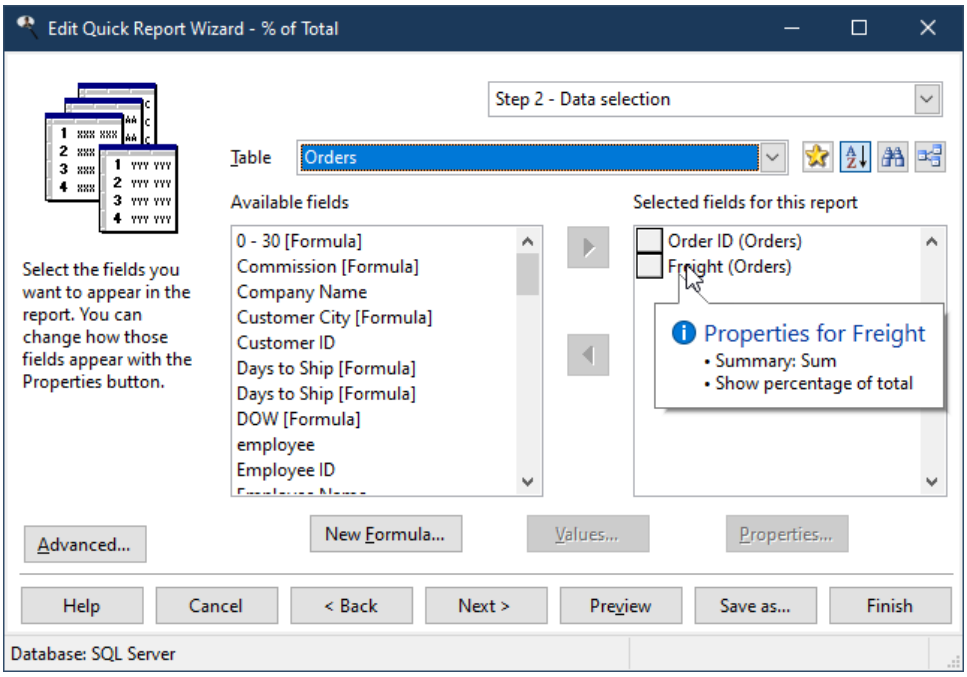

Figure 17. ctl32_BalloonTip allows you to display a more informative and attractive tooltip.

ctxTips provides a similar control to ctl32_BalloonTip. It can provide a simple rectangular window, a "thought bubble" window, a "circular bubble" window, or a "Vista tips" window with an optional title, body, and optional footer, each with its own image, fonts, and colors (**Figure 18**).

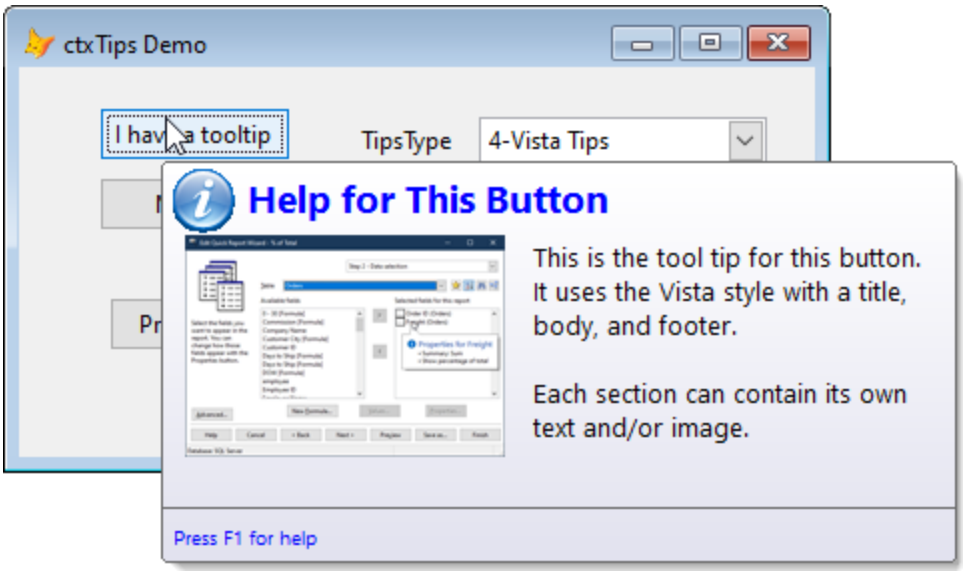

Figure 18. ctxTips can display large, attractive tooltip windows with text, images, titles, and footers.

Using ctxTips

Using ctxTips with VFP requires a little tweak over the ways it's used with other environments. The reason is because ctxTips expects each control has its own windows handle (hWnd) and displays a tooltip based on that. The first parameter passed to the AddTips and AddExtendedTip methods, which define a tooltip, is the hWnd of the control the tooltip is for. Since VFP controls don't have their own hWnd, instead we'll pass an ID value and then set the ItemOverID property of the ctxTips control to the appropriate ID in the MouseMove method of each control that has a tooltip. For example, the following code defines two tooltips, with IDs 1 and 2 respectively:

```
This.oTips.AddTips(1, 'This is the tool tip for button 1.')
This.oTips.AddTips(2, 'This is the tool tip for button 2.')
```

MouseMove for button 1 has this code:

```
lparameters tnButton, ;
   tnShift, ;
   tnXCoord, ;
   tnYCoord
Thisform.oTips.ItemOverID = 1
```

The code for button 2 is similar but sets ItemOverID to 2. MouseMove for the form sets it to 0, which clears the tooltip. Obviously, this is a pain to have to do manually for every control in every form, so you may wish to add a custom property to your base controls for the tip

ID, with a default of 0, and put code into MouseMove that sets ItemOverID to the value of the property.

To create a tooltip with a title, body, and image for the body, use AddEnhancedTips:

```
Thisform.oTips.AddEnhancedTips(1, 'This is the tool tip for this button.' + ccCRLF + ;
       'It uses the Vista style with a title,' + ccCRLF + ;
       'body, and footer.' + ccCRLF + ccCRLF + ;
       'Each section can contain its own' + ccCRLF + ;
       'text and/or image.', ;
       'Help for This Button', 1, 0)
```

The first parameter is the tooltip ID, the second is the body text, the third is the title text, the fourth is the body image number, and the last is the background image number.

Properties you'll likely set with ctxTips are:

- TipsType: this controls the type of tool tip, such a 1 for a thought bubble or 3 for a circular bubble. You'll probably want to use 4 for Vista tips, which gives the appearance shown in Figure 18.
- ControlStyle: set this to 1 to use a "Vista" style, with a background gradient. Alternatively, you can use your own gradient by setting FillType, BackColor, and BackColorTo to the desired values.
- Title*: these properties control whether the title is visible (TitleVisible), its font (TitleFont), foreground color (TitleForeColor), and so on. Its text can either be set with the TitleText property, which sets it for all tooltips, or using one of parameters in the AddEnhancedTips method, which makes it specific for that tooltip.
- Footer*: these properties control whether the footer is visible (FooterVisible), its font (FooterFont), foreground color (FooterForeColor), text (FooterText), and so on.
- TextAlign: if the tooltip uses multiple lines (separated with CHR(13) + CHR(10)), set this property to 0 for left justify, 1 for right, or 2 for center, as the default setting of 3 displays only a single line.
- ShadowBox: set it to .T. to display a shadow for the tooltip window.
- TextMargin: I find the default setting of 0 is too small, so I set it to 5.

Sample form

There are a couple of sample forms that come with the DBI controls. TipsDemo.scx (Figure 18) comes with this document and shows some more options ctxTips has.

As with ctxDropMenu, there's a 100 pixel wide, 16 pixel high grey shape at 0, 0 in the form, so don't put anything in that area or it'll be overlapped by that shape.

ctGroup

Years ago, I created a custom control that appears like a container with a labelled border. These are typically used to group related controls. It's a subclass of Container with a Shape and a Label on it. It's kind of a pain to use because after you drop one on a form, you have to drill into the container to set the Caption of the label and when you resize the container, you have to drill in and resize the shape to (this is only if you want it to appear properly at design time; at run time, it adjusts everything so it looks correct). I created a builder to help with that, but it's still a bit of work to get it just right.

ctGroup is a similar control but it's easy to use, especially since its Properties dialog gives you access to the most commonly changed properties. Set Caption to the caption for the label, BorderType to 5-Rounded, BackColor to the BackColor of the form, and Font to the desired font for the label (most of which you could of course do in a subclass and then use the subclass in your forms), then right-click, choose Edit, and add the controls to the container.

The Menu and Toolbar containers in Figure 12 are instances of ctGroup.

ctColorButton and ctColorCombo

If you allow your users to change the color of something (let's call it the target), you may have created a color picker as I did in the Project Explorer VFPX project (**Figure 19**): using a shape with FillStyle set to 0-Solid, FillColor set so it displays the current color (likely using code in Refresh), and a call to GETCOLOR() in the Click event to change both FillColor and the target color.

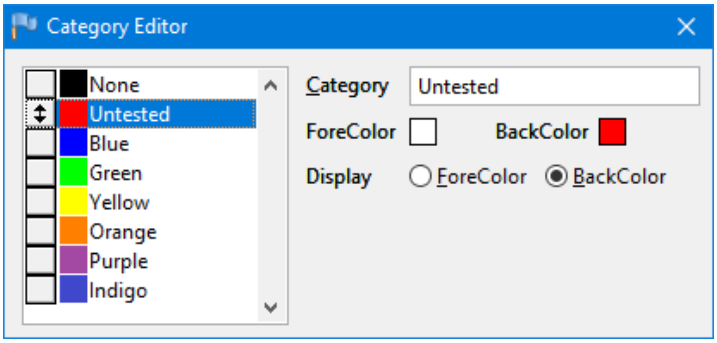

Figure 19. You can create simple color pickers using shapes like the ones for ForeColor and BackColor in the Category Editor of the Project Explorer.

For most of the samples that accompany this document, I used ctxColorButton instead. I subclassed ctxColorButton, added a cControlSource property that in an instance is set to the name of the target, and put the following code into the Refresh event to set the selected color to the desired value:

```
This.SelectColor = evaluate(This.cControlSource)
```

The ColorSelect event, which fires when the user chooses a color, has this code to set the color of the target:

```
lparameters tnColor
store tnColor to (This.cControlSource)
```

Figure 20 shows one of the sample forms with the color picker dropped down. Clicking the Other button displays a dialog similar to the one displayed when you call GETCOLOR().

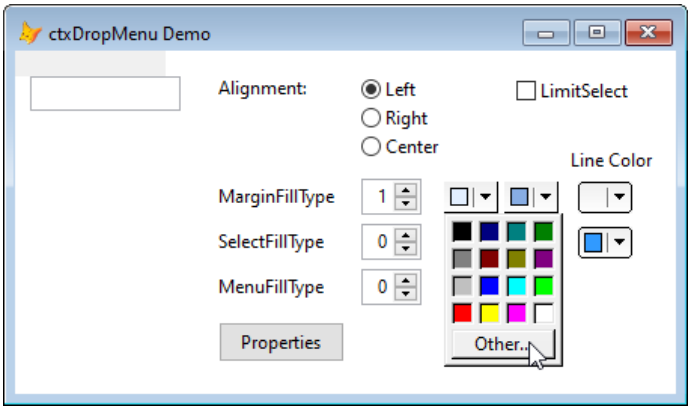

Figure 20. ctColorButton provides an easy way to create a color picker.

ctColorButton is filled with a default set of 16 colors but you can pick your own set by setting the Color1 through Color16 properties to the desired colors. As with other DBI controls, you have control over the appearance of the button through properties such as BackColor, BackColorTo, BorderColor, and so on.

ctColorCombo is similar to ctColorButton but displays colors in a dropdown list. If you set the StandardColors property to .T., it displays the same 16 colors ctColorButton does but if it's .F., you can call AddItem to add up to 32,000 colors to the list. You can also set the ItemHeight array elements for each item to the height of the color bar so you can provide a "line width" type of picker. In Refresh, set the Color property to the target color and in Change, set the target color to Color. You'll likely want to set ControlStyle to 5 for a modern appearance.

The ColorCombo.scx sample form (**Figure 21**) shows an example of using ctColorCombo to change the BorderColor and BorderWidth of a shape.

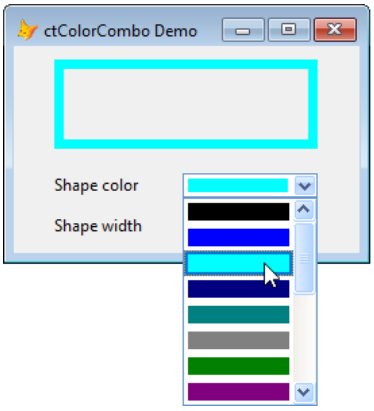

Figure 21. ctColorCombo can be used to select a color and/or a width from a dropdown list.

Other controls

Some other controls you may find useful are:

- ctxListView: a ListView can display its content as large icons, small icons, a list, or a report.
- ctxGauge: provides a customizable gauge control.
- ctxMEdit and ctxNEdit: similar to VFP TextBox but can optionally display spin and drop buttons (Medit is masked, NEdit is numerical).
- ctxSlide: a slider control.
- ctxMeter: a progress meter.
- ctxSplit: a splitter control. There isn't much advantage of this control over my SFSplitter control, available from my web site.
- ctFold: similar to a PageFrame but you can put images on the tabs.
- ctHyperLink: similar to the VFP HyperLink control but opens the user's default browser rather than Internet Explorer.
- ctExplorerBar: a Windows XP-era explorer bar. These aren't really used much anymore but you may find a use for it.
- ctxDate: a monthly calendar control (**Figure 22**).

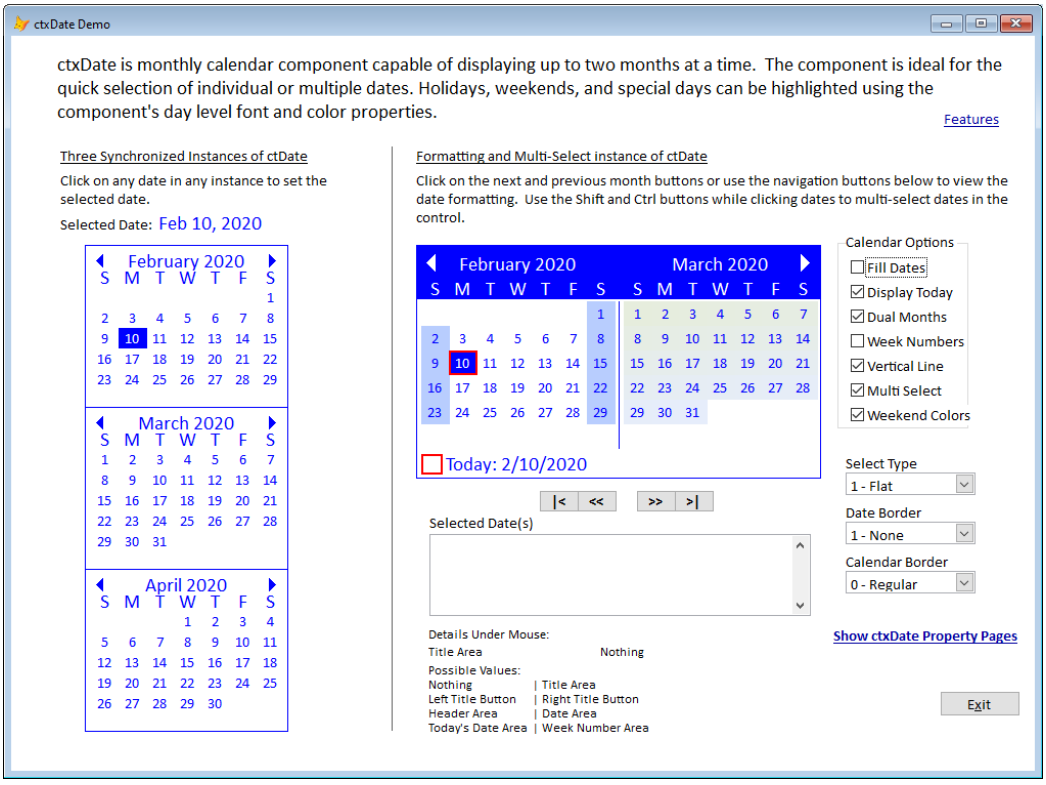

Figure 22. ctxDate provides a monthly calendar control.

- ctxCalendar: provides a highly customizable calendar control that can be used for scheduling (**Figure 23**).

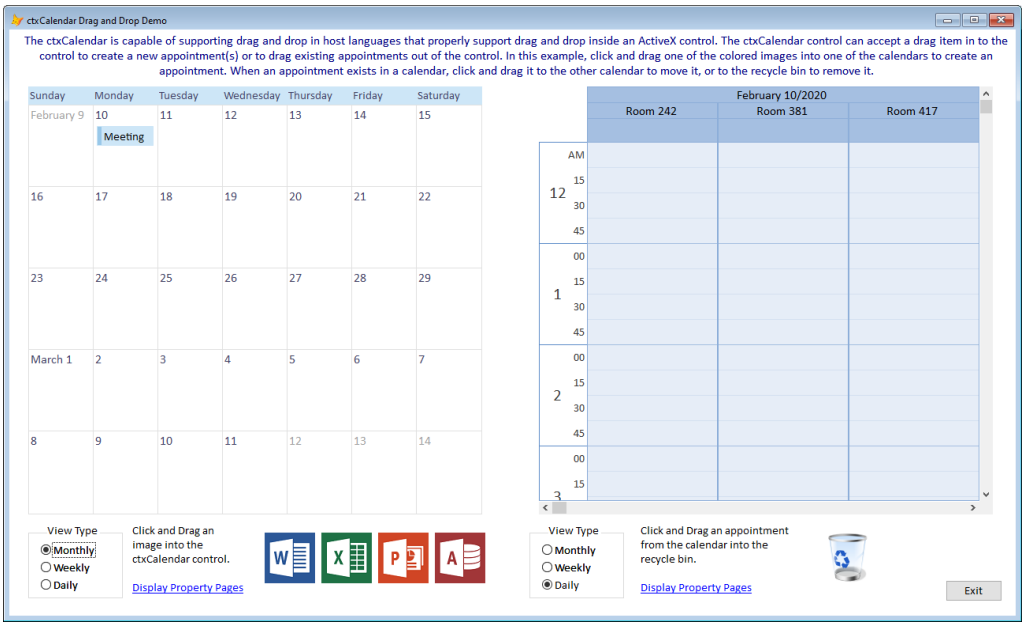

Figure 23. ctxCalendar can be used as a scheduling control.

- ctxYear: provides a calendar showing 3, 4, 6, or 12 months at a time (**Figure 24**).

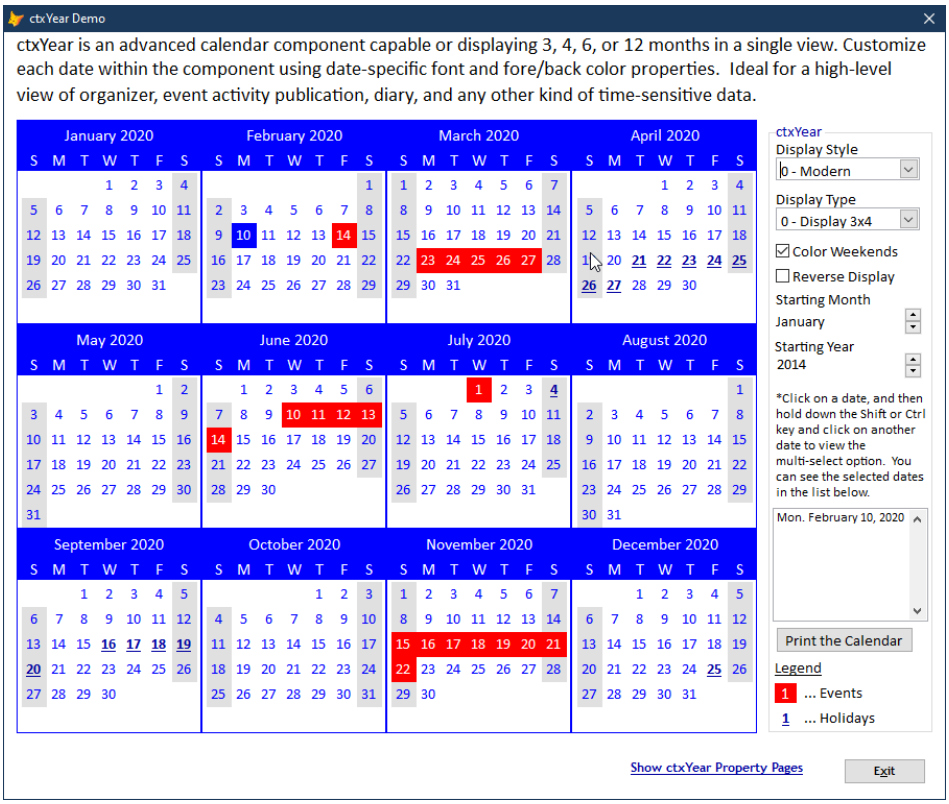

Figure 24. ctxYear provides a calendar showing 3, 4, 6, or 12 months at a time.

Run each of the sample forms that come with the DBI controls to get ideas about which ones might be useful to you.

Licensing and deployment

DBI controls are licensed per developer using the controls in a design surface. The run time components, however, are royalty-free. So, basically it's like VFP itself: you need to purchase one license per developer but distribution is free. DBI has single, five-developer, and site license pricing; see https://www.dbi-tech.com/Store_StudioControlsCOM.aspx for the 32-bit control prices and https://www.dbi-tech.com/Store_StudioControlsCOM64.aspx for the 64-bit prices.

To deploy an application using DBI controls, include the appropriate OCXs with your installer and register them. Each control is in its own OCX with a name matching the control (for example, the OCX for ctxTreeView is ctxTreeView.ocx) in the Studio Controls for COM\Components subdirectory of the folder where you installed the DBI controls. In addition to the OCX files, all DBI controls have dependencies on several Microsoft C DLLs; see the "What files do I need to include with distribution?" topic at https://tinyurl.com/ss7fyyu for details.

Summary

DBI's Studio Controls for COM allow you to modernize the appearance of your applications by providing attractive ActiveX controls that have more features than native VFP controls and the ActiveX controls that come with VFP. If you're using the 64-bit version of VFP Advanced, this is one of the only sets of controls available that is both tested and documented for VFP.

Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Query (https://stonefieldquery.com); the award-winning Stonefield Database Toolkit (SDT; now available as open source at https://github.com/DougHennig/StonefieldDatabaseToolkit); the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of *VFPX: Open Source Treasure for the VFP Developer*, *Making Sense of Sedna and SP2*, the *What's New in Visual FoxPro* series, *Visual FoxPro Best Practices For The Next Ten Years*, and *The Hacker's Guide to Visual FoxPro 7.0* (now available as open source at https://hackfox.github.io). He was the technical editor of *The Hacker's Guide to Visual FoxPro 6.0* and *The Fundamentals*. He wrote over 100 articles in 10 years for FoxRockX and FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe.

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (http://www.swfox.net) and 2020's

Virtual Fox Fest (https://virtualfoxfest.com). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.org) and has several projects available there. He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).