Session E-CTLX

Using ctl32 to Create a More Modern UI

Doug Hennig
Stonefield Software Inc.
2323 Broad Street
Regina, SK Canada S4P 1Y9
Email: dhennig@stonefield.com
Web site: www.stonefieldquery.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

Overview

Carlos Alloatti has created an incredible library called ctl32 that you can use to quickly add attractive controls to your VFP applications. It allows you to both provide a fresher, more modern interface and to add functionality that other VFP controls don't have. This document examines the library in detail and shows you how to implement the controls in your own applications.

Introduction

Many of my articles and conference sessions over the past couple of years have focused on controls that provide a more modern appearance to VFP applications. I continue that theme in this document by looking at the ctl32 library, created by Carlos Alloatti of Argentina.

ctl32 is a library of 22 controls that provide modern versions of some of the functions built into VFP or available as VFP or ActiveX controls. Included in the library are:

- Replacements for the VFP GETFILE(), PUTFILE(), and GETDIR() functions that use the latest dialogs available to your operating system.
- A replacement for the VFP status bar that has a lot more functionality.
- A scrollable container.
- A slider control.
- An object-oriented shortcut menu class.
- And many others.

I won't cover all of the controls, just those I think are the most useful. Specifically, we'll look at ctl32_DatePicker, ctl32_MonthCalendar, ctl32_BalloonTip, ctl32_ProgressBar, ctl32_OpenFileDialog, ctl32_StatusBar, ctl32_FormState, ctl32_Gripper, ctl32_ContextMenu, ctl32_Registry, ctl32_SContainer, and ctl32_TrackBar.

Download the ctl32 library from http://www.ctl32.com.ar and unzip it in some folder. Add the following files added to your project:

- ctl32.prg
- ctl32.vct and ctl32.vcx
- ctl32_api.prg
- ctl32_classes.prg
- ctl32_functions.prg
- ctl32_structures.prg
- ctl32_vfp2c32.prg
- vfpx.vct and vfpx.vcx

These files add about 1.5MB to your executable. That may seem like a lot, but you'll likely use many of the controls and replace ActiveX controls you formerly had to include with your application.

ctl32 comes with numerous samples, so I recommend spending some time going through them to see what's available and how the controls work. The ctl32 web site also has documentation on some of the controls.

Date picker

VFP comes with the Microsoft Date and Time Picker Control (DTPicker), an ActiveX control that provides a date control that appears like a combobox (Figure 1). You can type a date or click the down arrow to drop down a calendar control so the user can visually select the desired date. However, there are numerous shortcomings with this control:

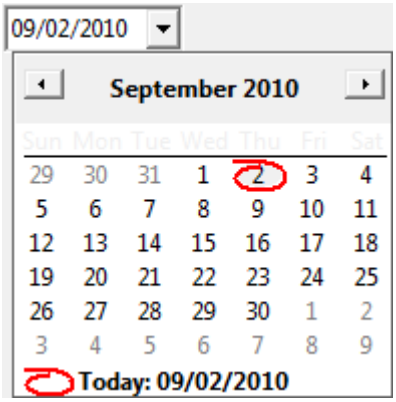

Figure 1. The Microsoft DTPicker.

- Since it's an ActiveX control, it's another file (MSComCtl2.OCX) you have to install and register.
- If you instantiate the control programmatically rather than visually (that is, you didn't drop it on a form or class), the user gets a license error when they run the form unless you specifically hack the Registry to install the license key for the control.
- It can't handle empty dates. The user has no way to blank the date and binding to a control source that has an empty value results in an error: "OLE IDispatch exception code 0 from DTPicker: A date was specified that does not fall within the MinDate and MaxDate properties... Unbinding object." You'll get this error when you run the sample TestDateCtrls form provided with the source code for this document.
- The combobox looks old: it has a 3-D sunken appearance rather than the flat appearance of modern controls.

Fortunately, ctl32 has a better control called ctl32_DatePicker. As you can see in Figure 2, it has a more modern appearance. Even better, it's written entirely in VFP so there are no ActiveX issues and source code is included so you can make any changes you wish.

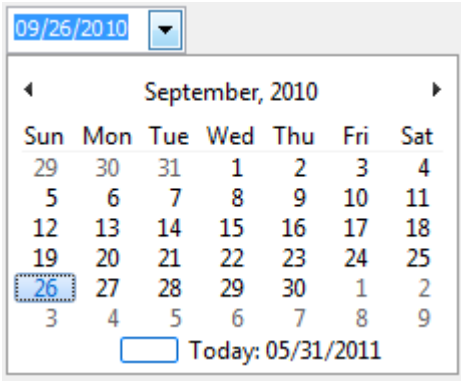

Figure 2. ctl32_DatePicker has a more modern appearance.

You can type a date, choose it from the drop-down calendar, or use one of the following keys:

- Up and down arrows (calendar closed): increment or decrement the selected part of the date. For example, if the cursor is in the month part, pressing the up arrow increments the month. If the entire date is selected, it increments or decrements the day. You can also use the mouse wheel to increment or decrement.
- Up, down, left, and right arrows (calendar open): moves the selected day around the calendar.
- Page up and down: increment or decrement the month.
- Ctrl+PgUp and Ctrl+PgDn: increment or decrement the year.
- Home: selects the first day of the month.
- End: selects the last day of the month.
- Ctrl+Home: selects today's date.

Clicking the month/year in the calendar title gives a nice animated effect: it "zooms out" to show the months for the specified year; click a month to display that month. Click the left or right arrows to move back or ahead a year. Click the year to zoom out again and show the years in a decade. Click again to show the decades in a century. This makes it much easier to visually select the desired year and month than the cumbersome mechanism the Microsoft control has.

Using ctl32_DatePicker is easy: simply drop it on a form, set ctlControlSource if you want to data bind it, set ctlUseDateTime to .F. if you're binding to a Date value or .T. for a DateTime value, and you're done. If you don't want to data bind the control, you can grab the selected date from ctlValue. If you want to specify the minimum and maximum dates the user can enter, set ctlMinDate and ctlMaxDate as desired. If you need to do something when the user changes the date, such as refreshing other controls on the form, put code in the ctlValueChanged method.

Month calendar

The calendar that appears when you click the down arrow in ctl32_DatePicker is actually another control, ctl32_MonthCalendar, you can use separately. ctl32_MonthCalendar has a number of properties that allow you to control its appearance. Some of these are:

- ctlCalendarDimensionsHeight and ctlCalendarDimensionsWidth specify the number of months to display vertically and horizontally.
- ctlTitleBackColor and ctlTitleForeColor specify the color for the calendar title. ctlTitleBackColor also specifies the font color for the days of the week.
- ctlTrailingForeColor specifies the color of the dates that precede and follow the displayed month.
- If ctlShowWeekNumbers is .T., the calendar displays week numbers at its left edge.
- Set ctlShowToday to .T. (the default) to display today's date at the bottom of the control.
- ctlMinDate and ctlMaxDate specify the minimum and maximum dates displayed.

The calendar can display certain dates in bold. Call ctlAddAnnuallyBoldedDate(dDate) to bold the same date every year, ctlAddMonthlyBoldedDate(dDate) to bold the same date every month, or ctlAddWeeklyBoldedDate(dDate) to bold the same date every week. Call ctlAddBoldedDate(dDate) to bold a single date.

To remove bolded dates, call ctlRemoveAnnuallyBoldedDate(dDate), ctlRemoveMonthlyBoldedDate(dDate), ctlRemoveWeeklyBoldedDate(dDate), or ctlRemoveBoldedDate(dDate). You can also call ctlRemoveAllAnnuallyBoldedDates(), ctlRemoveAllMonthlyBoldedDates(), ctlRemoveAllWeeklyBoldedDates(), or ctlRemoveAllBoldedDates() to remove all dates of the specified type.

Note that there's a bug in the ctlRemove*BoldedDates methods: after removing the specified date from an internal array, the code in all four methods redimensions the array without checking whether the new dimension is 0, which causes an error when you remove the last date. The fix is easy: in each method, surround the DIMENSION statement with a test. For example, here's the change I made in ctlRemoveBoldedDate:

```
if m.lnRows > 1
  Dimension This.ctlBoldedDates(m.lnRows - 1)
endif m.lnRows > 1
```

Figure 3 shows the sample TestMonthCalendar form, which demonstrates some of the features of ctl32_MonthCalendar.

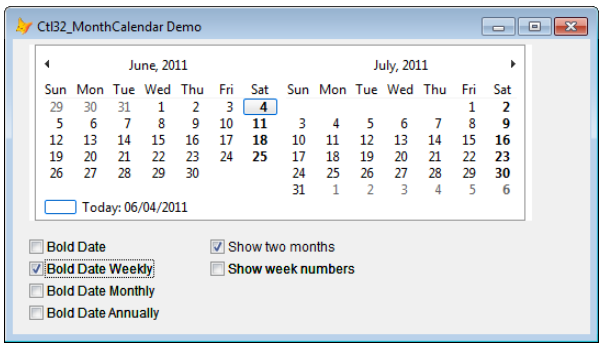

Figure 3. ctl32_MonthCalendar has many options to control its appearance.

Balloon Tips

The tooltips VFP displays (which are actually defined by Windows, not VFP) are kind of boring looking. As you can see in Figure 4, the tooltip window is tiny, with a yellow background, and provides no control over its appearance other than the text. For a more attractive appearance and much better control, try using ctl32_BalloonTip.

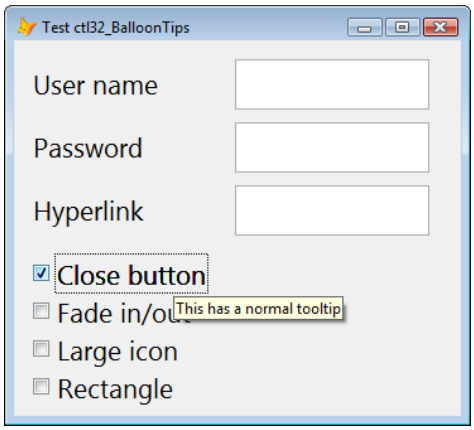

Figure 4. VFP's tooltips are kind of boring looking.

ctl32_BalloonTip displays tooltips such as that in Figure 5. Note the tooltip window has a "word balloon" appearance, a gradient background, a title, an icon, and of course, the tooltip text.

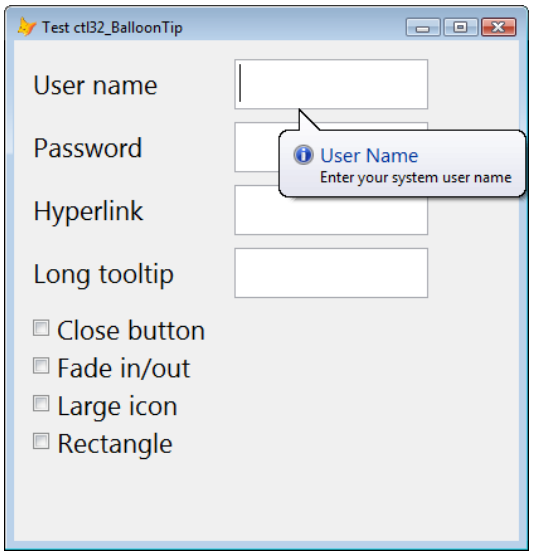

Figure 5. ctl32_BalloonTip displays a more attractive tooltip.

Start by dropping an instance of ctl32_BalloonTip on a form. Even though individual controls will have different tooltips, there can only be one instance of ctl32_BalloonTip per form. When a control that should display a balloon tip receives focus, call the ctl32_BalloonTip's ctlShow method with the appropriate parameters to display the balloon. Similarly, when the control loses focus, call ctlShow(0) to hide the balloon. If you want balloon tips displayed when the mouse moves over the control, regardless of whether it has focus or not, call ctlShow.

To make it easy to use balloon tips for textboxes, I created a subclass of Textbox called SFTextBox (in Samples.VCX) with the following changes:

- I added custom cBalloonTipTitle (the title of the balloon tip), lHaveBalloonTipControl (set to .T. in Init if the form has a balloon tip control), and nBalloonTipIcon (the icon to use for the balloon tip) properties.
- Init sets lHaveBalloonTipControl if the form has an object named oBalloonTip, which is assumed to be an instance of ctl32_BalloonTip.
- I added two custom methods: ShowBalloonTip and HideBalloonTip. HideBalloonTip simply calls Thisform.oBalloonTip.ctlShow(0) to hide the balloon tip. ShowBalloonTip has the following code to display the balloon tip:

```
lparameters tlGotFocus
#define CON_BTPOS_ACTIVECTRL  2
#define CON_BTPOS_MOUSE       6
with Thisform.oBalloonTip
    do case
        case empty(This.cBalloonTipTitle) or ;
            not This.lHaveBalloonTipControl
        case not empty(This.PasswordChar)
            .ctlCapsLockStyle = .T.
        otherwise
            .ctlShow(iif(tlGotFocus, ;
                CON_BTPOS_ACTIVECTRL, ;
                CON_BTPOS_MOUSE), ;
                This.ToolTipText, ;
                This.cBalloonTipTitle, ;
                This.nBalloonTipIcon)
    endcase
endwith
```

This code sets ctlCapsLockStyle to .T. if this is a password textbox, which causes a special balloon tip to appear if the Caps Lock key is turned on (Figure 6). Otherwise, it displays a balloon tip with the appropriate information.

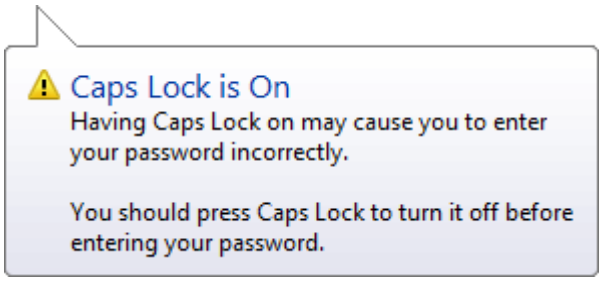

Figure 6. Setting ctlCapsLockStyle to .T. displays this balloon tip when the Caps Lock key is turned on.

- GotFocus and MouseEnter call ShowBalloonTip, although GotFocus passes .T. to indicate that we're receiving focus rather than the mouse moving over the control. The code in ShowBalloonTip puts the balloon tip over the control if passed .T. and near the mouse if passed .F.
- LostFocus and MouseLeave call HideBalloonTip.

To use SFTextBox, first drop a ctl32_BalloonTip control on a form and name it oBalloonTip. Then drop an SFTextBox on a form, set TooltipText to the text to display, cBalloonTipTitle to the title, and nBalloonTipIcon to the icon to use. If you want to change how a balloon tip appears, such as using a close button or large icons, set the appropriate properties of oBalloonTip. If you want to change something about this control's balloon tip, such as specifying which icon to use, override the code in ShowBalloonTip. For example, this code shows how to add a hyperlink to a balloon tip and how to specify a different icon:

```
#define TTI_ERROR      3
```

```
lparameters tlGotFocus
local lcTitle, lcText, lcURL
with Thisform.oBalloonTip
    lcTitle = 'ctl32_BalloonTip Documentation'
    lcURL = 'http://www.ctl32.com.ar/ctl32_balloontip_members.asp'
    lcText = .ctlMakeLink(lcTitle, lcURL)
    This.ToolTipText = 'This displays a hyperlink. Check ' + lcText + ' for details.'
    This.nBalloonTipIcon = TTI_ERROR
endwith
dodefault(tlGotFocus)
return
```

There are lots of other properties you can set to control the appearance of the balloon tip, including its background color, text font and size, position, margins, and so on. http://tinyurl.com/2dg4md8 has complete documentation on the properties and methods for ctl32_BalloonTip. Some you might find useful are:

- ctlCloseButton: set this to .T. to display a close button in the balloon tip window.
- ctlFadeIn and ctlFadeOut: set these to .T. to have the balloon tip window fade in and fade out.
- ctlIconLarge: set this to .T. to display a large version of the specified icon.
- ctlActive: set this to .F. to disable all balloon tips (you might do this for more experienced users, for example).
- ctlStyle: set this to 2 to display a rectangle rather than the default balloon shape (1).

Progress Bar

For several years, I used the Microsoft ProgressBar control to show the progress of some action. However, like the Microsoft DateTime control I discussed earlier, this is an ActiveX control, so it has the same issues as the DateTime control. And like the DateTime control, it looks dated (see the top progress bar in Figure 7).

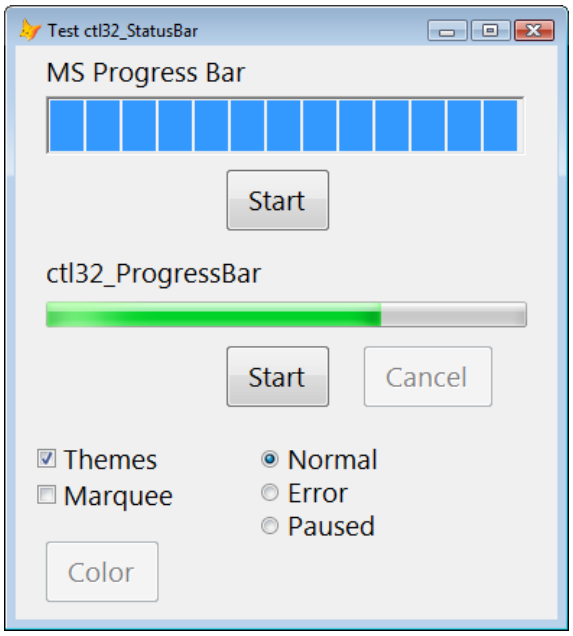

Figure 7. The ctl32_ProgressBar control is more attractive and flexible than the Microsoft ProgressBar control.

Fortunately, the ctl32 library includes ctl32_ProgressBar, a very attractive, modern- looking progress bar that also has more flexibility than the Microsoft control (see the second progress bar in Figure 7).

http://tinyurl.com/2v9vty7 has documentation for the properties and methods of ctl32_ProgressBar. I'll only discuss the more commonly used properties here.

Using ctl32_ProgressBar is very straightforward:

- Drop an instance on a form.

- Set some properties as necessary. If the range of values isn't 0 to 100, set ctlMinimum and ctlMaximum to the desired range. If you don't wanted a themed progress bar (I'm not sure why you wouldn't), set ctlThemes to .F. and other properties for non-themed progress bars, such as ctlBorderColor, ctlForeColor, ctlBackColor, and so on. If you want the bar to be red or yellow, set ctlState to 2 or 3, respectively.
- To indicate that something is happening, set ctlValue to the desired value. The progress bar displays a bar of the appropriate length.
- If you want to show a "marquee" effect (a block continuously scrolls across the progress bar without you changing ctlValue), set ctlMarquee to .T.
- If you want to know the current percentage that ctlValue is of ctlMaximum, use ctlPercent (a numeric value) or ctlValuePercent (a character string formatted with a %).
- If you run the sample TestProgressBar form, you'll notice something interesting: the progress bar length lags behind the percentage slightly. That's because the progress bar actually uses animation to draw the bar to the correct length. For example, if you click the "Set to 50" button, you'll notice that the bar doesn't jump to 50 but instead grows towards it.

File dialogs

VFP developers have relied on GETFILE(), PUTFILE(), GETDIR(), and GETPICT() for years to display file and directory selection dialogs. However, these dialogs have several shortcomings:

- You don't have much control over them: you can specify the file extensions, the title bar caption, and a few other options, but these functions hide most of the settings available in the native dialogs. For example, you can't specify a starting folder or default filename for GETFILE().
- These functions return file and folder names in upper-case rather than the case the user entered or the file actually uses.
- GETFILE() returns only a single filename even though the native dialogs optionally support selecting multiple files.

The most important issue, however, is that the dialogs displayed are from the Windows XP era and don't support new features added in Vista and Windows 7. Figure 8 shows the dialog presented by GETFILE(). Although this dialog does have a Recent Places button, it still looks like a dialog from an older operating system.

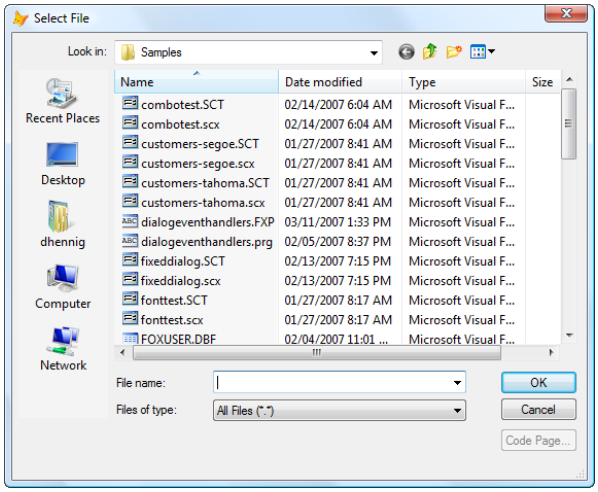

Figure 8. The VFP GETFILE() function is an older-looking dialog.

As you can see in Figure 9, the Windows 7 open file dialog not only has a more modern interface, it has several features the older dialog doesn't, including back and forward buttons, the "breadcrumb" folder control, and access to Windows Search.

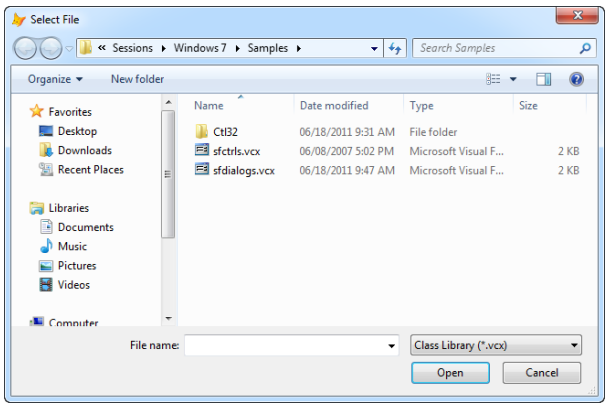

Figure 9. The Windows 7 open file dialog looks modern and has features the older dialog doesn't.

ctl32 has several classes that replace these functions, providing more control over the dialogs and using the latest dialogs available to your operating system, meaning they work correctly whether you're using Windows XP or Windows 7. The classes of interest here are ctl32_OpenFileDialog, ctl32_SaveFileDialog, ctl32_OpenPictDialog, and ctl32_FolderBrowserDialog. You can find documentation for these classes at http://www.ctl32.com.ar/ctl32_filedialog.asp.

ctl32_OpenFileDialog and ctl32_SaveFileDialog are subclasses of ctl32_FileDialog; they don't have any code, just properties set so they behave as their names suggest. You can either use these classes or use ctl32_FileDialog directly and set the properties as you wish. ctl32_OpenPictDialog is a subclass of ctl32_OpenFileDialog but has no changes from the parent class so it doesn't really act like GETPICT() unless you manually specify the file types allowed.

Rather than using ctl32_OpenFileDialog and ctl32_SaveFileDialog directly, I created a wrapper class for them called SFCommonDialog in SFDialogs.VCX. This class has the public properties shown in Table 1.

Table 1. Properties of SFCommonDialog.

Property	Description
aFileNames[1]	An array of filenames returned from the dialog
cDefaultExtension	The default file extension to display
cFileName	The name of file selected or initially set as default
cFilePath	The path files were selected from
cFileTitle	The file title property of the selected file(s)
cInitialDirectory	The initial directory to show files from
cTitlebarText	The caption for dialog title bar
lAllowMultiSelect	.T. to allow selection of multiple files
lFileMustExist	.T. if the file must exist
lHideReadOnly	.T. to hide read-only files from list
lNewExplorer	.T. to use the "new" explorer user interface and features such as the Places bar
lNoPlacesBar	.T. to not include Places bar in the dialog
lNoValidate	.T. to not validate the file name
lOverwritePrompt	.T. to prompt to overwrite if the user selects an existing file
lPathMustExist	.T. if the path must exist
lSaveDialog	.T. to use the Save dialog instead of the Open one
nFileCount	The number of files selected from the dialog
nFilterIndex	Specifies which of the filters the user selected from the dialog or the default filter to use

SFCommonDialog has three methods: AddFilter(cDescription, cFilter), which adds file type filters, ClearFilters(lClearAll), which clears the file type filters, and ShowDialog, which displays the dialog and returns .T. if the user clicked OK.

Here's an example, taken from Dialogs.SCX, that displays a Save dialog; this code produces the dialog shown in Figure 9. This code specifies that VCX and all files (the default) should be available. It sets nFilterIndex to 2 to force VCX files as the default file type, and lOverwritePrompt to .T. so the user receives a warning if they select an existing file.

```
with Thisform.oDialog
    .AddFilter('Class Library (*.vcx)', '*.vcx')
    .nFilterIndex      = 2
    .lSaveDialog       = .T.
    .cTitlebarText     = 'Select File'
    .cDefaultExtension = 'vcx'
    .lOverwritePrompt  = .T.
    if .ShowDialog()
        lcFile = 'You chose ' + .cFileTitle + ' from ' + .cFilePath
        messagebox(lcFile)
    endif .ShowDialog()
endwith
```

ctl32_FolderBrowserDialog appears to be unfinished: several options, such as ctlShowNewFolderButton, don't do anything (the code that uses them in the _GetFlags method is commented out) and the class needs a couple of fixes before it can be used. In ctlShowDialog, comment out the ?m.lcPath statement, since you likely don't want the path the user selected output to the screen or active form, and change the return statement to return m.lcPath instead of 0. The dialog that appears when you call ctlShowDialog isn't appreciably different from GETDIR(), but this class does have the benefit of returning the selected folder in the correct case.

Status bar

Many applications use a status bar at the bottom of their main forms to display status information, such as the current state, location of the cursor, current page number, printing progress, and so on. For example, Figure 10 shows the status bar for Microsoft Word.

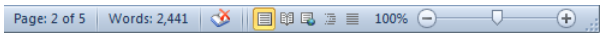

Figure 10. The status bar for Microsoft Word provides information about the application.

The native VFP status bar isn't often used in production applications for several reasons:

- You have little control over it other than the text specified in SET MESSAGE.
- It's really intended as an IDE status bar because it displays information of interest to developers, such as the path of the current alias and the current record number.
- It looks old, with sunken panels and MS Sans Serif for text.
- It only appears in _SCREEN, which means it doesn't appear at all if you turn off _SCREEN and use a top-level form instead.

I've used Carlo's ctl32_StatusBar (Figure 11) to display a status bar in my applications for several years and like it a lot. It provides some very nice features, including a built-in progress bar, and allows you to specify which panels appear, what size they are, and what text and icon to use. See http://www.ctl32.com.ar/ctl32_statusbar.asp for documentation.

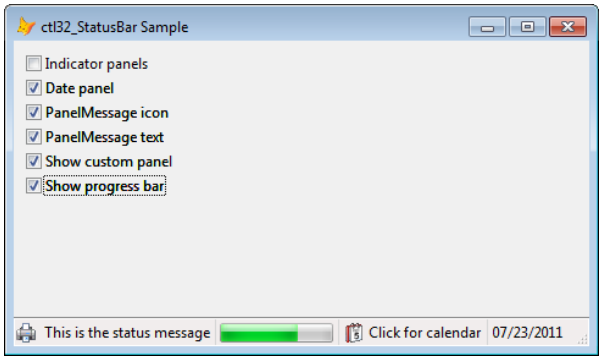

Figure 11. ctl32_StatusBar is a full-featured status bar.

ctl32_StatusBar consists of several built-in panels, all but the first of which can be hidden:

- The first panel displays the same messages the native VFP status bar does, such as the status bar text for menu items and controls, but not the path or record number for the current alias. However, you can specify your own text to display by setting the ctlMessage property, which overrides the VFP status bar text. You can also optionally specify an icon. You can reference this panel using the PanelMessage property (for example, set PanelMessage.ctlIcon to specify an icon and PanelMessage.ctlAlignment to specify the alignment for the panel).
- The second panel contains a ctl32_ProgressBar control. Reference it through the ProgressBar member of the status bar. It's hidden by default so set ProgressBar.ctlVisible to .T. when you want it displayed.
- The next set of panels are custom ones you define. By default, there are five custom panels (ctlPanelCount controls this). You can change this value at design time but not at runtime in code. Reference them through the ctlPanels array, such as setting the text for the first custom panel using ctlPanels(1).ctlCaption.
- The next three panels display the state of the Insert, CapsLock, and NumLock keys. Reference these via PanelOvr, PanelCaps, and PanelNum.
- The last panel displays the current date, in the format defined by the regional settings for your system. You can specify how the date is displayed by setting PanelDate.ctlFormat to 0 (not visible), 1 (short date), 2 (long date), or 8 (month and year).

Some of the properties you can set for the panels are shown in Table 2.

Table 2. Panel properties.

Property	Description
ctlAlignment	0 for left, 1 for right, or 2 for center alignment.
ctlAutoSize	Set this to .T. to autosize the panel (its width is based on the content) or .F. to set the width to ctlWidth.
ctlCaption	The text in the panel.
ctlIcon	The path and name for a 16x16 ICO file to display in the panel.
ctlVisible	Set to .T. to make the panel visible or .F. for hidden.
ctlWidth	The width of the panel if ctlAutoSize is .F.

You can also set properties of the status bar itself. ctlCaption is a shortcut for PanelMessage.ctlCaption: it sets the text for the first panel. Similarly, ctlIcon is a shortcut for PanelMessage.ctlIcon. Set ctlSizeGrip to .T. if you want a "gripper" displayed in the lower right corner of the status bar.

ctl32_StatusBar has events for a click (ctlClick), right-click (ctlRightClick), and double-click (ctlDblClick). You can either put code into these events or BINDEVENT to them. The nXCoord and nYCoord contain the X and Y coordinates of the mouse pointer, relative to the upper-left corner of the status bar and nPanel indicates what panel the user clicked:

- -2: the user clicked a border or the gripper
- -1: PanelMessage
- 0: ProgressBar panel but not the progress bar itself
- 101: PanelOvr
- 102: PanelCaps
- 103: PanelNum
- 104: PanelDate
- Value between 1 and 100: the custom panel with that number

To use ctl32_StatusBar, drop it on a form. It doesn't matter where you place it; at runtime, it automatically moves to the bottom of the form and sizes itself as the form is resized. One quirk is that the status bar automatically

takes the bottom 24 pixels of the form, so it'll cover any controls you put near the bottom. I like to move the status bar to the bottom of the form at design time so I can see exactly how much space it'll take at run time.

The sample form accompanying this document, StatusBar.SCX, shows a few of the features of ctl32_StatusBar. Notice that if PanelMessage Text is unchecked, the status bar displays the status bar text of the selected control or menu item. Click one of the panels to display a message about where you clicked.

Form state

One sign of a professional, polished application is that forms open at the same position and size as they were when closed. It means the user doesn't have to constantly resize and reposition windows to their preferred location.

This seemingly simple task—saving the form's Top, Left, Height, and Width properties in Destroy and restoring them in Init—is actually more complicated than it seems. What if there are no saved values (that is, this is the first time the form has been opened)? What if the form was minimized or maximized the last time? What if the user lowered the screen resolution since the last time? What if the form was positioned on a second monitor and that monitor isn't connected now?

Fortunately, Carlos has created a class named ctl32_FormState that deals with these issues. All you have to do is drop it on a form and it automatically handles all state saving and restoring tasks. Of course, it also provides some control over how it works; see Table 3 for some of the properties you can set. You can also call ctlClearState to clear the saved state of the form.

The StatusBar sample form accompanying this document has a ctl32_FormState control on it so it demonstrates the user of this control. Move the form and resize it, then close and rerun the form to see how it works. The samples that come with ctl32 (ctl32_FormState_*.SCX in the Example subdirectory) show various uses of the control, including a demonstration of how the form is restored when it's off-screen when closed.

Table 3. ctl32_FormState properties.

Property	Description
ctlAutoCenter	If .T., ignores the saved Top and Left settings and instead auto-centers the form.
ctlRestoreMaxState	Set this to .T. to restore a maximized form to that state.
ctlRestoreMinState	Set this to .T. to restore a minimized form to that state.
ctlRestoreMinPos	When you minimize a form, it appears as just a title bar in the lower left corner of the screen. You can move this title bar somewhere else. Set ctlRestoreMinPos to .T. to restore the position of the title bar.
ctlRestoreStateinIDE	Set this to .F. to only restore state when running in an EXE, not in the IDE.
ctlRestoreToPrimary	Set this to .T. to restore the form to the primary monitor if it was last located on a monitor that no longer exists.
ctlRegKeyName	ctl32_FormState saves form state data in the Windows Registry in the FormState key of HKEY_CURRENT_USER\Software*ExeName*, where ExeName is the name of the EXE ("VFP9" when running in the IDE). ctlRegKeyName allows you to specify the key name.

Gripper control

Some applications display a gripper control in the lower-right corner as a visual indication that the window is resizable. The dots in the corner of Figure 10 show what Word's gripper looks like.

VFP forms don't natively have a gripper control. If you use ctl32_StatusBar, you get one for free. For other forms, you can add ctl32_Gripper. Drop one somewhere on a form—it automatically positions itself in the lower-right corner at runtime—and set ctlStyle to the type of gripper you want to display: 1 to use the style for the current operating system, 2 for lines (Windows 2000 style), or 3 for dots. You can also set ctlScale to the scale you wish to use; for example, 2.0 means draw it at twice normal size.

The user can now resize a form by dragging either the lower-right corner of the form or dragging the gripper control to size the form as desired.

Context menu

One of the last places in VFP that isn't object-oriented is the menu system. This makes it awkward to create a flexible menu that, for example, uses different languages or removes certain items for certain users. Fortunately, there are several OOP-based menus, including a couple of VFPX projects (http://vfpx.codeplex.com).

Carlos also has an OOP menu, specifically for shortcut rather than system menus, in ctl32_ContextMenu. In addition to providing an OOP menu, this class uses native-drawn menus rather than VFP-drawn menus, so they look like the shortcut menus you see in other applications rather than the older-style menus typical of VFP applications.

ctl32_ContextMenu is simple and easy to use:

- Drop an instance on a form.
- Call Clear to clear any existing menu in case the menu was displayed before.
- Call Add to add menu items to the menu.
- Call Show to display the menu and get the selected item, typically from the RightClick method of a control.

The first call to Add creates the menu, so just pass it the key you want assigned to the menu. Subsequent calls to Add create menu items in the menu, so pass the menu key, the key to assign to the menu item, the caption for the item, and optionally the name of an image to use in the item. To add items as a submenu of another item, pass the key for the item as the first parameter.

For example, the following code, taken from the RightClick method of cmdSubmenu in ContextMenu.SCX included with the source code for this document, creates a menu with three items and three subitems for the third item and displays the menu:

```
with Thisform.oContextMenu

* Clear the menu.

    .Clear()

* Create the main popup menu.

    .Add('MainMenu')

* Add menu items.

    .Add('MainMenu', 'opt1', 'Item 1')
    .Add('MainMenu', 'opt2', 'Item 2')
    .Add('MainMenu', 'opt3', 'Item 3')

* Add submenu items to item 3.

    .Add('opt3', 'opt4', 'Item 4')
    .Add('opt3', 'opt5', 'Item 5')
    .Add('opt3', 'opt6', 'Item 6')

* Show the menu at the mouse position.

    .Show()
endwith
```

Run the form and click the Submenu button to see the results shown in Figure 12.

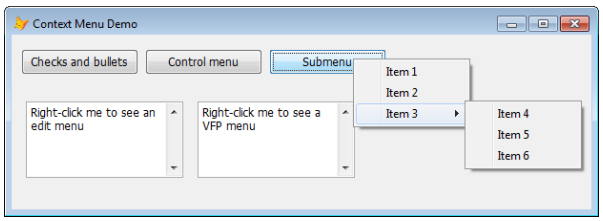

Figure 12. ctl32_ContextMenu created this shortcut menu with submenu.

If you pass no parameters to Show, it displays the menu at the current mouse position. Otherwise, pass the X and Y coordinates for the menu.

After you've added menu items to the menu, you can set additional properties of the items before calling Show. Use the MenuItems collection of ctl32_ContextMenu to access these items by key. Some of the properties you can set are:

- Picture: this sets the image of the item to the specified file name, like passing the fourth parameter to Add does. Due to what appears to be a bug in ctl32_ContextMenu, I couldn't get images to display, either when specified in Add or by setting Picture.
- Enabled: set this to .F. to disable the item. The last menu item in Figure 13 has Enabled set to .F.
- Caption: this is the caption, which you can also set by passing the caption to use as the third parameter to Add.
- Checked: set this to .T. to display a checkmark in front of the item. Click the Checks and Bullets button in the sample form to see an example; the second set of three items can be turned on or off (Figure 13).
- RadioCheck: set this to .T. for all items that should behave as option buttons. That is, one of them, which has Checked set to .T., displays a bullet in front of it. Choosing a different item clears the bullet from the former one and adds it to the selected one. In Figure 13, only one of the first three items has a bullet.
- DefaultItem: set this to .T. to have the item appear bolded. The first item in Figure 13 has this turned on.

The ShowFlags property provides the ability to animate the drawing of the menu. See http://tinyurl.com/3cwymg5 for a list of the choices. Ctl32.H provides constants, such as TPM_VERPOSANIMATION, you can use for these values. I must admit, the animation must be very fast or very subtle on my system because I couldn't see much difference setting ShowFlags to different values.

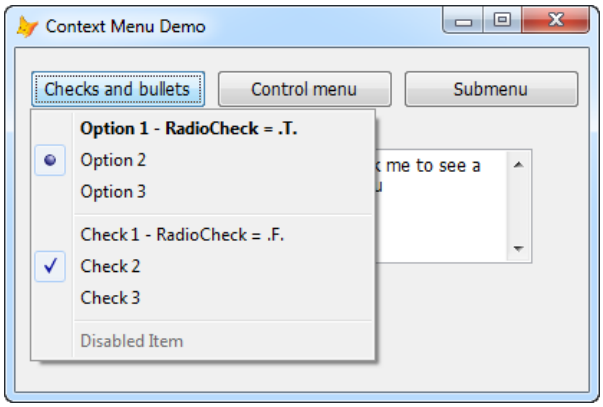

Figure 13. The Checks and Bullets button shows a menu with different menu options.

Ctl32_ContextMenu has a couple of methods that display built-in menus. ShowEditMenu displays a menu with the usual Cut, Copy, Paste, and other edit menu items. Right-click the left editbox in the form to see this menu. Right-click the right editbox to see a typical VFP shortcut menu. ShowControlMenu displays the same menu displayed when you right-click the title bar of a window; click the Control Menu button in the sample form to see an example.

Registry

VFP's FFC folder includes Registry.VCX, which provides classes to work with the Windows Registry: reading from and writing to keys, deleting keys, and enumerating keys. However, it's a little odd to work with: most of the methods return a code indicating whether the method succeeded or not rather than the value you're actually interested in, so you have to pass variables to contain return values by reference.

ctl32_Registry is a much easier class to use. Call SetValue(cName, uValue) to save a value to the Registry. cName can include a complete path under the current "hive" (the default is HKEY_CURRENT_USER), such as "Software\FoxRockX\NameOfValue", or can be just the value name if you've set the RegistrySubkey property to the subkey to write to. For example, this code:

```
loReg.SetValue('Software\FoxRockX\String', 'This is a string')
```

does the same as this:

```
loReg.RegistrySubkey = 'Software\FoxRockX'
loReg.SetValue('String', 'This is a string')
```

To specify a different hive, set RegistryKey to the name of the hive, such as "HKEY_LOCAL_MACHINE." Note than in Windows Vista and later, HKEY_LOCAL_MACHINE is a protected resource and writes to it are actually redirected to a different location.

SetValue can handle more than just strings, which is another benefit over the FFC class. The following works just fine:

```
loReg.SetValue('DateTime', datetime())
loReg.SetValue('Logical', .T.)
```

To read a value from the Registry, use GetValue(cName, uDefaultValue). uDefaultValue is the value to return if the key doesn't exist, but GetValue also uses its data type to determine how to cast the value it reads from the Registry. For example, here's how to read the values saved in the previous examples:

```
lcValue = loReg.GetValue('String', '')
ldValue = loReg.GetValue('Date', {/})
ltValue = loReg.GetValue('DateTime', {/:})
llValue = loReg.GetValue('Logical', .F.)
```

GetValues and SetValues are interesting methods: pass them a reference to a container object, such as a Form or Page in a PageFrame, and the subkey to use, and they read from and write to the Registry the Value property of every control in the container. This is a fast way to save and restore the contents of the controls in a form.

If you want to determine if a key exists, call SeekValue(cName); it returns .T. if the key can be found.

DeleteValue(cName) deletes the specified value in the Registry. DeleteSubkey(cSubkey) deletes the specified subkey. However, it doesn't delete the subkey if it contains any subkeys; for that, use DeleteSubkeyTree(cSubkey).

Registry.PRG in the source code for this document demonstrates the use of ctl32_Registry.

The one shortcoming of ctl32_Registry is that it doesn't support enumerating keys.

Scrolling container

ctl32_SContainer is a very cool control: it allows you to create a container with scroll bars (vertical, horizontal, or both) so the container can contain a larger set of controls than can be displayed at once.

Figure 14 shows a sample form that comes with ctl32, ctl32_SContainer_01.SCX. At 640 by 480, the image in this form is much larger than the ctl32_SContainer it's contained in, so the scroll bars allow you adjust the viewing area and see other parts of the image. Figure 15 shows the form after the image has been scrolled to the lower-right corner.

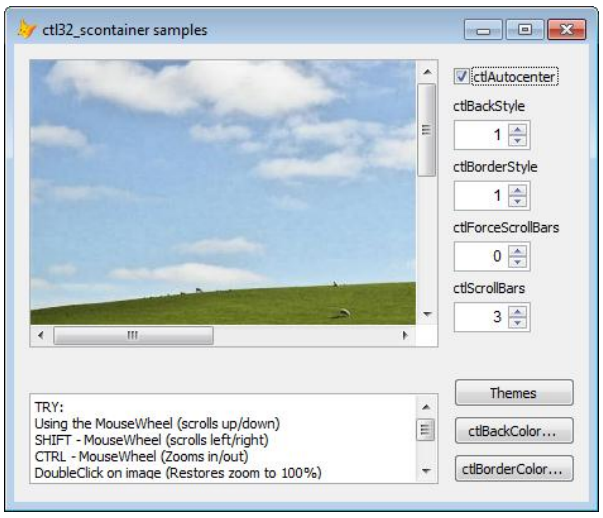

Figure 14. The ctl32_SContainer in this form is scrolled to the upper-left corner of the image.

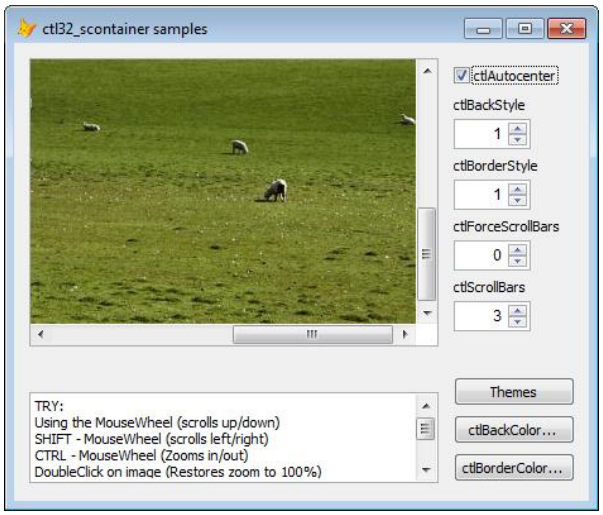

Figure 15. The image is scrolled to the lower-right corner.

The benefit of using ctl32_SContainer is that your form can contain a large set of controls while the form itself is relatively small. For example, another sample form, ctl32_SContainer_05.SCX, displays 16 large icons in a scrolling panel. The form would have to be wider than the screen to display these images without scrolling.

Using ctl32_SContainer is easy. Drop one on a form, add controls to it like you would any other container, and then set properties specifying its behavior. Some of the properties of interest are:

- ctlBorderStyle: set this to 0 for no border for the container or 1 to have a border.
- ctlScrollBars: set this to 0 for no scroll bars, 1 for horizontal, 2 for vertical, or 3 for both. The user can scroll the container using the scroll bars or by dragging inside the container. You can also programmatically set ctlVValue and ctlHValue to the desired scroll positions.
- ctlForceScrollBars: if this is the default of 0, scroll bars only appear when the contents of the container aren't completely visible. Set it to 1 to force scroll bars to always appear.
- ctlBorderColor: set this to the desired border color.
- ctlBackColor and ctlBackStyle: these work like the normal BackColor and BackStyle properties for controls. The background color only appears if BackStyle is 1 and the container is larger than the controls it contains.
- ctlAutoCenter: if this is .T., the controls are centered within the container (which isn't apparent unless the container is larger than the controls).
- ctlAllowZoom: if ctlAllowZoom is .T., holding down the Ctrl key while moving the mouse wheel zooms the image in and out, and double-clicking restores the size to 100%. You can also call ctlRestoreSize to restore the zoom level programmatically.

For a practical example of ctl32_SContainer, download http://tinyurl.com/3kxe38b. This is the Southwest Fox registration application created by Rick Schummer. He used this control to allow a large set of controls to fit inside a reasonably-sized form.

Slider control

VFP ships with several ActiveX controls, one of which is the Microsoft Slider Control. This control is sometimes used in place of a spinner control when you want the user to change a numeric value in discrete steps. Like the other ActiveX controls, the Slider control has a few downsides: it's an additional OCX file you have to deploy and register, it doesn't come with source code so it can't be customized, and it looks like a 1990's era control.

Contrast the top slider with the bottom one in Figure 16. The bottom one, an instance of ctl32_TrackBar is a more modern looking control. As part of ctl32.VCX, it comes with source code and there's nothing else to deploy. You also have more control over the appearance and behavior of the control.

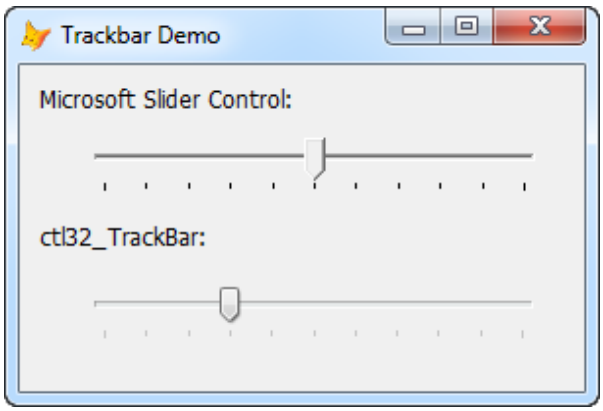

Figure 16. ctl32_TrackBar is a more modern looking slider control than the ActiveX version that comes with VFP.

To use ctl32_TrackBar, drop it on a form and set some properties:

- ctlValue is the value of the slider. It's set automatically by moving the thumb along the track but you can also set it programmatically to a starting value.
- Rather than manually reading from and writing to ctlValue, set ctlControlSource to the desired control source.
- ctlMinimum and ctlMaximum specify the lower and upper values the control can have.
- ctlLargeChange contains the amount to increment or decrement ctlValue by when you click the track on either side of the thumb or press Page Up or Page Down.
- ctlSmallChange is similar but determines the amount to change ctlValue by when the left and right arrows are pressed.
- ctlTickFrequency determines the spacing between the tick marks on the track.
- Set ctlOrientation to 0 for a horizontal slider or 1 for vertical.
- ctlTickStyle specifies how tick marks should appear: 0 for none, 1 for above (horizontal slider) or to the left (vertical slider) of the track, 2 for below or to the right, and 3 for both sides. When it's set to 3, the thumb appears as a rounded rectangle rather than a pointed one.
- Set ctlShowFocusCues to 1 to draw a dotted border around the control when it has focus or 0 for no border.
- ctlThumbSize sets the size of the thumb in pixels. 0 means use the default size.

Summary

Carlos Alloatti has created an incredible library you can use to quickly add attractive controls to your VFP applications. It allows you to both provide a fresher, more modern interface and to add functionality that other VFP controls don't have. I've used several of Carlos' controls for years now and love how easy they are to use and how they make my applications work and look better.

Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; 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. He was the technical editor of The Hacker's Guide to Visual FoxPro 6.0 and The Fundamentals. All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

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). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). 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).