Mine for Code in XSource

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Web site: http://www.stonefield.com
Web site: http://www.stonefieldquery.com

Overview

Visual FoxPro comes with source code for most of the "Xbase" tools that ship with the product, including the Class Browser, Code References, Toolbox, and Task Pane. Looking at source code written by top VFP gurus often gives insight into new, powerful coding techniques. This document looks at various files in XSource to show you cool ideas you can incorporate and even better, code you can directly use in your applications today.

Introduction

Many of the tools that come with VFP were actually written in VFP rather than C++ (the language used to create VFP itself). These are often referred to as “Xbase” tools to distinguish them from internal components. Some of the Xbase tools are available from the Tools menus, such as the Class Browser, IntelliSense Manager, Toolbox, Task Pane manager, and Report Wizard. Other are available from the appropriate places, such as the Referential Integrity Builder, CursorAdapter Builder, and in VFP 9, ReportBuilder.APP, which provides the new dialogs for the Report Designer.

For several versions now, VFP has shipped with the source code for these XBase tools. Because there’s a lot of code, it comes as a ZIP file: XSource.ZIP in the Tools\XSource subdirectory of the VFP home directory. Unzipping this file results in a subdirectory of Tools\XSource called VFPSource which contains the source code for the XBase tools. **Table 1** lists the purpose of each directory in XSource and the location of the APP file generated from this source.

Table 1. The purpose and location of applications created with XSource source code.

XSource Directory	Purpose	Location
AddLabel	AddLabel application	Tools\AddLabel\AddLabel.APP
Beautify	Beautify tools	Beautify.APP
Browser	Class Browser and Component Gallery	Browser.APP and Gallery.APP
Builders	Main builder program (delegates to appropriate builder APP) and main wizard program (delegates to appropriate wizard APP)	Builder.APP and Wizard.APP
Builders\Builders	Common files used in builders	-
Builders\CmdBldr	CommandGroup Builder	Wizards\CmdBldr.APP
Builders\EditBldr	EditBox Builder	Wizards\EditBldr.APP
Builders\FormBldr	Form Builder	Wizards\FormBldr.APP
Builders\GridBldr	Grid Builder	Wizards\GridBldr.APP
Builders\ListBldr	ListBox Builder	Wizards\ListBldr.APP
Builders\RIBuildr	Referential Integrity Builder	Wizards\RIBuildr.APP
Builders\StylBldr	Style Builder	Wizards\StylBldr.APP
Convert	Converter (converts files to newer versions)	Convert.APP
Coverage	Coverage Profiler	Coverage.APP
DataExplorer (VFP 9)	Data Explorer (available in Task Pane Manager)	DataExplorer.APP
EnvMgr	Environment Manager (available in Task Pane Manager)	EnvMgr.APP
FoxCode	IntelliSense Manager	FoxCode.APP
FoxRef	Code References	FoxRef.APP
MemberData (VFP 9)	MemberData Editor	MemberDataEditor.APP
OBrowser	Object Browser	ObjectBrowser.APP
ReportBuilder (VFP 9)	Dialogs and other behavior for Report Designer	ReportBuilder.APP
ReportOutput (VFP 9)	Report listeners (e.g. XML and HTML)	ReportOutput.APP
ReportPreview (VFP 9)	New report preview dialog	ReportPreview.APP
TaskList	Task List	TaskList.APP
TaskPane	Task Pane Manager	TaskPane.APP
ToolBox	ToolBox	ToolBox.APP
VFPClean	Utility to restore VFP user files and registry settings	VFPClean.APP
WebService	Web Services Publisher	Wizards\WebService.APP
Wizards\AnchorEditor (VFP 9)	Anchor Editor	Wizards\AnchorEditor.APP
Wizards\Automate	Common files used in Graph, Mail Merge, and PivotTable Wizards	-
Wizards\DEBuilder	CursorAdapter and DataEnvironment Builders	Wizards\DEBuilder.APP
Wizards\WZApp	Application Wizard	Wizards\WZApp.APP
Wizards\WZCommon	Common files used in wizards and builders	-
Wizards\WZForm	Form Wizard	Wizards\WZForm.APP
Wizards\WZFoxDoc	Documenting Wizard	Wizards\WZFoxDoc.APP
Wizards\WZGraph	Graph Wizard	Wizards\WZGraph.APP
Wizards\WZImport	Import Wizard	Wizards\WZImport.APP
Wizards\WZIntNet	WWW Search Page Wizard	No longer included as tool
Wizards\WZMail	Mail Merge Wizard	Wizards\WZMail.APP
Wizards\WZPivot	PivotTable Wizard	Wizards\WZPivot.APP
Wizards\WZQuery	Query Wizard	Wizards\WZQuery.APP
Wizards\WZReport	Report and Label Wizards	Wizards\WZReport.APP
Wizards\WZTable	Table and Database Wizards	Wizards\WZTable.APP and Wizards\WZDBC.APP

Wizards\WZUpsize	Upsizing Wizard	Wizards\WZUpsize.APP
Wizards\WZWeb	Web Publishing Wizard	Wizards\WZWeb.APP

Note: some other subdirectories of Tools—Analyzer, CPZero, Filer, GenDBC, HexEdit, MSAA, and Test—contain source code for the tools in those directories. These tools aren't accessible from menus in VFP; they're standalone tools available by running the appropriate file in the appropriate directory. I won't discuss these tools in this document.

What's so good about having the source for these tools? At least two things:

- If one of these tools doesn't do exactly what you want, you can change whatever you need to and build a new APP file.
- I always like to leverage good work done by others, and the VFP Xbase tools were written by some of the best and brightest stars in the VFP universe (many were written by non-Microsoft employees, such as Markus Egger, who wrote the Object Browser). There are very interesting techniques used in some of these tools, and I've learned a lot by spelunking through the source code. More importantly, some of the classes, images, and PRGs are useful in other applications. That use is the basis for this document.

If you decide to use some of the XSource source code in your own applications, be aware that some classes are subclasses of classes in different VCX files, and some of the VCXs have a lot of classes in them. Even if you only need one of them, all the related VCXs, PRGs, and image files will come along for the ride when you build your project. You can either live with it (the only effect is the EXE size, and disk space is cheap these days) or pull the classes you want out of the VCX into your own smaller VCX. Tools such as the Class Browser and White Light Computing's HackCX (a tool I highly recommend; http://www.whitelightcomputing.com) can help with this, but it still might be a bit of a chore to get it right.

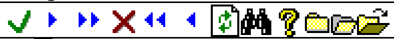
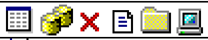
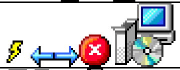
This document doesn't explore everything in XSource; that would take a whole book. Instead, I'll discuss certain things I think are useful, including:

- How the Toolbox's Outlook-like bar works.
- How to use the FOXUSER resource file as an alternative to INI files or the Windows Registry.
- Displaying AVI files in VFP forms.
- Creating object-oriented shortcut menus.

Rich source of images

When I'm looking for images for buttons or icons for forms, one of the first places I look is XSource, especially the directories for the newer XBase tools (those added in VFP 8 or later). I often find exactly what I'm looking for, or something close than I just need to tweak a bit in an editor such as Paint. For example, **Table 2** shows some of the useful images in XSource directories.

Table 2. Some useful images in XSource directories.

XSource Directory	Images
Browser	
Coverage	
DataExplorer	
FoxRef	
OBrowser	
ReportPreview	
TaskList	
TaskPane	

Toolbox	

Saving settings in the resource file

A professionally-written application saves certain settings so the next time the user runs it, those same settings are used again. For example, it's nice when an application "remembers" the size and position of forms, the location where you last imported a file from, what type of file you last exported to, and so forth. There are several places an application can store such settings, including INI files, the Windows Registry, an XML file, or a table.

The VFP interactive development environment (IDE) does a great job of remembering things such as the size, position, and selected text in code files, and the last opened code window in a form or class. It saves these settings in your resource file, typically FoxUser.DBF in your HOME(7) directory. Wouldn't it be nice to use the resource file for your own settings?

That's exactly what the FoxResource class in FoxResource.PRG does. This class, found in several XSource directories including Toolbox, provides methods to read from and write to a resource file. It doesn't have to be the current resource file (that is, the one returned by SYS(2005)); you can specify the name of the file to use in the ResourceFile property.

A resource file consists of the following fields:

- TYPE: this contains the type of record. Most records have "PREFW" in this field, but you can use any value you wish by setting the ResourceType property of FoxResource (it defaults to "PREFW").
- ID: the ID for the record. This can be any value you wish, but it should be something unique.
- NAME: the name for the record. This can be left blank if you wish.
- READONLY: .T. if the record shouldn't be changed.
- CKVAL: the checksum for the DATA field.
- DATA: the settings. FoxResource saves your settings as an array using SAVE TO MEMO DATA.
- UPDATED: the date the record was last updated.

FoxResource loads a set of settings from a record in the resource file using the Load method. Pass the method the ID and the name (matching the ID and NAME fields in the file) and it will load the settings, if such a record exists, into its internal collection of name-value items. Once you've loaded a set, you can retrieve values using the Get method; pass it the name of the setting you want to retrieve. You don't have to worry about data types; since the settings are saved as an array in the resource file, the values come back with the same data type as they were saved with. This is a nice change from INI files, for example, where everything is a string. If the setting you request doesn't exist, Get returns .NULL., so you probably should use the NVL() function on the return value. Here's an example that leaves Top alone if the setting doesn't exist:

```
This.Top = nvl(oFoxResource.Get('Top'), This.Top)
```

You can check if a setting exists using the OptionExists method. Pass it the name of the setting and it returns .T. if the setting exists.

To store a setting, call the Set method, passing it the name and value of the setting. Once you've stored a set of settings, you can save the entire set to the resource file with the Save method. This method expects the same ID and name parameters as the Load method. Save supports read-only records in the resource file; it refuses to update the record if the READONLY field is .T. It also updates the DATE and CKVAL columns.

There are a few other, less useful, methods. Clear clears the collection, GetData returns the contents of the DATA memo field for the specified record, SaveTo saves to a specified memo field rather than the DATA memo, and RestoreFrom restores from the specified memo field.

TestMenu.SCX demonstrates the use of FoxResource. The Init method instantiates FoxResource, loads the setting for the TESTMENU name, calls SetFormPosition to restore the form size and position, then uses FoxResource's Get method to restore the record pointer and filter.

```
local lnRecno, ;
  lcFilter
with This
  .oResourceOptions = newobject('FoxResource', ;
    home() + 'Tools\XSource\VFPSource\Toolbox\FoxResource.prg')
  .oResourceOptions.Load('TESTMENU')
  .SetFormPosition()
  lnRecno = nvl(.oResourceOptions.Get('RecNo'), 0)
  if between(lnRecno, 1, reccount())
    go lnRecno
  endif between(lnRecno, 1, reccount())
  lcFilter = nvl(.oResourceOptions.Get('Filter'), '')
  .SetFilter(lcFilter)
endwith
```

Speaking of shamelessly lifting, oops, I mean, leveraging, code, the code for the SetFormPosition method comes straight from the same named method in the cFoxForm class in ToolboxCtrls.VCX in the Toolbox source directory. This code does a little more than simply restoring the Top, Left, Width, and Height properties for the form; it also ensures that these values don't cause the form to be off-screen (for example, if the form was originally placed far to the right on a high-resolution system and now is being opened on a lower-resolution display).

```
LOCAL nTop, nLeft, nWidth, nHeight

IF !ISNULL(THIS.oResourceOptions)
  m.nHeight = THIS.oResourceOptions.Get("HEIGHT")
  IF VARTYPE(m.nHeight) == 'N' AND m.nHeight >= 0
    THIS.Height = m.nHeight
  ENDIF
  m.nWidth = THIS.oResourceOptions.Get("WIDTH")
  IF VARTYPE(m.nWidth) == 'N' AND m.nWidth >= 0
    THIS.Width = m.nWidth
  ENDIF

  m.nTop = THIS.oResourceOptions.Get("TOP")
  IF VARTYPE(m.nTop) == 'N'
    * make sure we're visible
    IF !THIS.Desktop
      IF m.nTop > _SCREEN.Height
        m.nTop = MAX(_SCREEN.Height - THIS.Height, 0)
      ELSE
        IF m.nTop + THIS.Height < 0
          m.nTop = 0
        ENDIF
      ENDIF
    ENDIF

    THIS.Top = m.nTop
  ENDIF

  m.nLeft = THIS.oResourceOptions.Get("LEFT")
  IF VARTYPE(m.nLeft) == 'N'
    * make sure we're visible
    IF !THIS.Desktop
      IF m.nLeft > _SCREEN.Width
        m.nLeft = MAX(_SCREEN.Width - THIS.Width, 0)
      ELSE
        IF m.nLeft + THIS.Width < 0
          m.nLeft = 0
        ENDIF
      ENDIF
    ENDIF

    THIS.Left = m.nLeft
  ENDIF
ENDIF
```

The Destroy method saves the settings we want preserved and then writes them all to the TESTMENU set in the resource file.

```
with This
  .oResourceOptions.Set('Left',   .Left)
  .oResourceOptions.Set('Top',    .Top)
  .oResourceOptions.Set('Height', .Height)
  .oResourceOptions.Set('Width',  .Width)
  .oResourceOptions.Set('Filter', .cFilterName)
  .oResourceOptions.Set('RecNo',  recno())
  .oResourceOptions.Save('TESTMENU')
endwith
```

To try this out, run TestMenu.SCX and move the form anywhere on the screen. Right-click on the form and choose a filter setting from the Set Filter submenu, then right-click and choose Next several times. (We'll discuss the how the shortcut menu was created next.) Close the form and then re-run it; it should open in the same position with the same filter and same record displayed when you close it.

Object-oriented shortcut menus

VFP's menu system is one of the few things that remains procedural rather than object-based. While you can create your own menus using the various menu-related commands (such as DEFINE POPUP and DEFINE BAR), almost no one does because VFP includes the Menu Designer which allows us to create menus visually. However, the biggest downside to using menus generated by the Menu Designer is that they can't be altered once they're defined. That means you can't easily localize them or show or hide specific bars under certain conditions.

ContextMenu to the rescue! This class, defined in FoxMenu.PRG in the Toolbox source directory, allows you to create a shortcut menu on the fly without having to write ugly DEFINE POPUP and DEFINE BAR commands. Simply instantiate the class, set up the bars as desired (which could use conditional code to localize the menu or decide which bars to include), and tell it to display the menu. You could even data-drive the menu if you wish.

To add bars to the menu, call the AddMenu method. This method accepts up to six parameters (the last four are optional): the prompt for the bar, an expression to execute when the bar is selected, the name of an image file to use for the bar's picture, .T. if the bar's checkmark is turned on, .T. if the bar is enabled, and .T. if the bar appears in bold. Because VFP's menu system lives outside its object system, references like This or Thisform won't work in the expression to execute, so put a reference to the form or object into a private variable and use that variable in the expression. The sample code for this section illustrates this.

AddMenu returns a reference to an object containing properties about the menu bar: Caption, Picture, Checked, ActionCode, IsEnabled, and Bold. More interestingly, however, it also contains a SubMenu property, which contains a reference to another ContextMenu object. So, to create a submenu for a bar, simply call the AddMenu method of the bar's SubMenu object.

To display the menu, call the Show method. In the VFP 8 version, you can optionally pass two parameters: the row and column at which the menu should be displayed. The VFP 9 version adds a third optional parameter: the name of the form the menu should appear in (ContextMenu uses the IN WINDOW clause of the DEFINE POPUP command in this case). Don't call Show with no parameters or the menu isn't quite placed in the right spot. Instead, pass either MROW('') and MCOL('') for the row and column and omit the form name or omit the row and column and pass This.Name for the form name.

A bug in the BuildMenu method of the VFP 9 version of this class prevents this from working quite right. Add a semi-colon at the end of the first line shown below and uncomment the second.

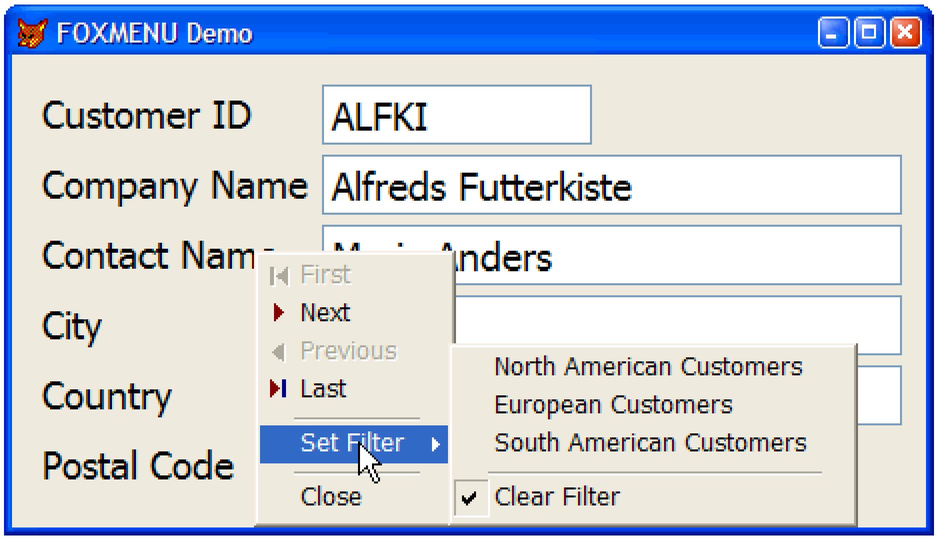
```
DEFINE POPUP (m.cMenuName) SHORTCUT RELATIVE FROM m.nRow, m.nCol
  * IN WINDOW (m.cFormName)
```

ContextMenu has a couple of properties. MenuBarCount contains the number of bars in the menu. ShowInScreen is supposed to indicate that the menu is displayed in _SCREEN, but all code referencing that property is commented out, so you can ignore it.

Here's an example, taken from the ShowMenu method of TestMenu.SCX (see **Figure 1**). It instantiates the ContextMenu class and calls its AddMenu method to create the various bars to display. The code creating the navigation bars (First, Next, Previous, and Last) passes the picture and enabled parameters so the bars have images and are enabled only when appropriate. The Set Filter bar has a submenu of different types of filters that can be set, and the bar matching the current filter has its checkmark turned on.

```
local loMenu, ;
  loMenuItem
```

```
private poForm

* Create the menu object.

loMenu = newobject('ContextMenu', ;
  home() + 'Tools\XSource\VFPSource\Toolbox\FoxMenu.prg')

* Create a private reference to this form so the menu actions will work.

poForm = This

* Define the menu bars.

loMenu.AddMenu('First',    'poForm.FirstRecord()',    'frsrec_s.bmp', .F., ;
  not This.lFirstRecord)
loMenu.AddMenu('Next',     'poForm.NextRecord()',     'nxtrec_s.bmp', .F., ;
  not This.lLastRecord)
loMenu.AddMenu('Previous', 'poForm.PreviousRecord()', 'prvrec_s.bmp', .F., ;
  not This.lFirstRecord)
loMenu.AddMenu('Last',     'poForm.LastRecord()',     'lstrec_s.bmp', .F., ;
  not This.lLastRecord)
loMenu.AddMenu('\-')
loMenuItem = loMenu.AddMenu('Set Filter')
loMenuItem.SubMenu.AddMenu('North American Customers', ;
  'poForm.SetFilter("NA")', '', This.cFilterName = 'NA')
loMenuItem.SubMenu.AddMenu('European Customers', ;
  'poForm.SetFilter("EU")', '', This.cFilterName = 'EU')
loMenuItem.SubMenu.AddMenu('South American Customers', ;
  'poForm.SetFilter("SA")', '', This.cFilterName = 'SA')
loMenuItem.SubMenu.AddMenu('\-')
loMenuItem.SubMenu.AddMenu('Clear Filter', ;
  'poForm.SetFilter("")', '', empty(This.cFilterName))
loMenu.AddMenu('\-')
loMenu.AddMenu('Close', 'poForm.Release()')

* Display the menu.

loMenu.Show(, , This.Name)
```


FOXMENU Demo

Customer ID: ALFKI
Company Name: Alfreds Futterkiste
Contact Name: Maria Anders
City
Country
Postal Code

Figure 1. This shortcut menu, complete with a submenu, was created with the ContextMenu class and just a few lines of code.

Displaying video files

When you delete or move a lot of files, Windows Explorer displays a nice dialog showing an animated version of what it's doing. It doesn't make the process any faster, but at least you know your system isn't hung up while it's doing its job.

For several versions now, VFP has shipped with animation files in the Graphics\Videos subdirectory. You can use these videos to present the same type of progress dialog Windows Explorer does. The cAnimation class in FoxRef.VCX (in the FoxRef XSource directory) makes it easy to do that. Simply drop one of these on a form and call its Load method, passing it the name of the video file to play. If you want to load the file but not play it right away, set lAutoPlay to .F., call Load to load the file, and then call Play when you're ready to play the video. cAnimation is a subclass of the Microsoft Animation Control, so you'll need to distribute this ActiveX control (MSCOMCT2.OCX) with your application.

To make it even easier to use, I created a subclass of cAnimation called SFAnimation in SFXSource.VCX. The subclass has a cFileName property that contains the name of the video file, and its Init method calls Load, passing it This.cFileName. So, simply drop an SFAnimation control on a form, set cFileName, and you're done.

Run TestAnimation.SCX to see an example of this control. It plays six of the video files that come with VFP, displaying animations similar to what you see in Windows Explorer progress dialogs. **Figure 2** shows what this form looks like when running.

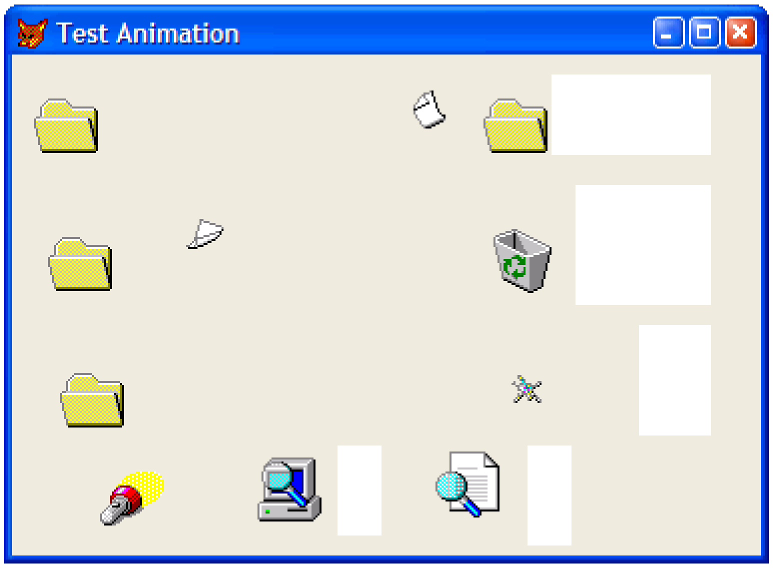


Figure 2. The cAnimation and SFAnimation classes make it easy to display video clips in VFP forms.

Enumerating SQL Server instances

Occasionally, you may encounter the need to determine whether SQL Server is available and if so, the server name. Doing so requires using Windows API functions or SQL-DMO calls, neither of which are much fun to code. Fortunately, the Data Explorer, a new application that comes with VFP 9 (it's available in the Task Pane Manager) already has code to do this.

The SQLDatabaseMgmt class in DataMgmt_SQL.PRG in the DataExplorer XSource directory provides a wrapper around Windows API functions needed to enumerate the SQL Servers available on a network, as well as the databases in a specific server, the tables, views, and stored procedures in a specific database, and other useful bits of information. It's very easy to use: instantiate the class and call its GetAvailableServers method to retrieve a collection of servers. To retrieve information about a particular server, call the Connect method to connect to the server, then the desired method. For example, GetDatabases returns a collection of databases for the server. Here's some sample code, adapted from DisplayServers.PRG:

```
loSQL = newobject('SQLDatabaseMgmt', 'DataMgmt_SQL.prg')

* Get a collection of available servers and display the databases for each
* one, then display the tables for the first database.

loServers = loSQL.GetAvailableServers()
for each loServer in loServers
  if loSQL.Connect(loServer.Name)
    messagebox('Connected to server ' + loServer.Name)
    loDatabases = DisplayDatabases(loSQL, loServer.Name)
    if loDatabases.Count > 0
      loDatabase = loDatabases.Item(1)
      DisplayTables(loSQL, loDatabase.Name)
    endif loDatabases.Count > 0
  else
    messagebox('Could not connect to ' + loServer.Name)
  endif loSQL.Connect(loServer.Name)
next loServer
set path to

* Display the databases in the current server.

function DisplayDatabases(toSQL, tcServer)
local loDatabases, ;
  lcDatabases, ;
  loDatabase
loDatabases = toSQL.GetDatabases()
if loDatabases.Count > 0
  lcDatabases = ''
  for each loDatabase in loDatabases
    lcDatabases = lcDatabases + iif(empty(lcDatabases), '', chr(13)) + ;
      loDatabase.Name + ' (' + loDatabase.Owner + ')'
  next loDatabase
  messagebox(lcDatabases, 0, 'Databases in ' + tcServer)
else
  messagebox('Could not retrieve databases for ' + tcServer + chr(13) + ;
    chr(13) + toSQL.LastError)
endif loDatabases.Count > 0
return loDatabases

* Display the tables in the specified database.

function DisplayTables(toSQL, tcDatabase)
local loTables, ;
  lcTables, ;
  loTable
toSQL.DatabaseName = tcDatabase
loTables = toSQL.GetTables()
if loTables.Count > 0
  lcTables = ''
  for each loTable in loTables
    lcTables = lcTables + iif(empty(lcTables), '', chr(13)) + ;
      loTable.Name + ' (' + loTable.Owner + ')'
  next loTable
  messagebox(lcTables, 0, 'Tables in ' + tcDatabase)
else
  messagebox('Could not retrieve tables for ' + tcDatabase + chr(13) + ;
    chr(13) + toSQL.LastError)
endif loTables.Count > 0
return
```

There are similar classes (they all share a common parent class) in the same directory that work with Oracle, VFP, SQL-DMO, and ADO data sources. The overhead is light: just DataMgmt.PRG (which contains the parent class) and DataMgmt_SQL.PRG (or whichever specific subclass you wish to work with).

Progress dialog

Displaying an animated video doesn’t always give the user enough information about the progress of a long-running process. Many times, the user wants to see what step a multi-step process is at, or a thermometer indicating how far along the process is. The cProgressForm class in FoxToolbox.VCX in the Toolbox XSource directory provides a nice progress dialog you can use for such a task. (There’s also a version of this class in FoxRef.VCX in the FoxRef XSource directory, but the Toolbox version has a few more capabilities.)

As you can see in **Figure 3**, cProgressForm provides several features:

- an animated image—the computer with the magnifying glass—indicating something is happening (this isn’t a video file; it’s an animated GIF)
- a description of the overall process

- a thermometer indicating the progress of the process
- a status message showing what step the process is currently on
- a Cancel button, allowing the user to terminate the process.

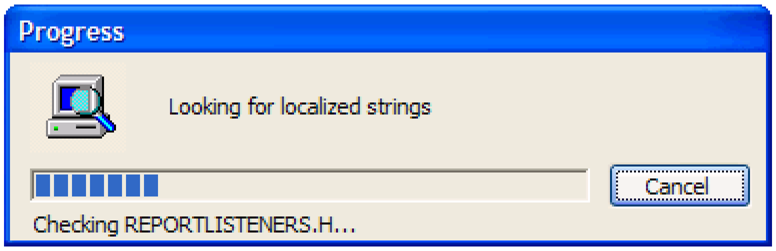


Figure 3. The cProgressForm class provides a nice progress dialog.

When you instantiate cProgressForm, you can optionally pass it the description of the process (such as "Retrieving data") to set the Caption of the description label. You can also set the description by calling the SetDescription method. As your process progresses, you may want to update the Caption of the status label below the thermometer to indicate what step you're on; pass the desired string to the SetStatus method. If you want the thermometer to show absolute values, such as the current record number out of the maximum number of records, rather than the percentage of completion, set the maximum value for the thermometer by calling SetMax. If you don't want the Cancel button displayed, set the lShowCancel property to .F. If you don't want the thermometer displayed, set the lShowThermometer property to .F.

To update the thermometer, call the SetProgress method, passing the value for the thermometer (either as an absolute value if you called SetMax or as a percentage value, such as 50 when the process is half done) and optionally a string to use as the Caption for the status label. SetProgress returns .T. if the process should continue or .F. if it should stop because the user pressed the Cancel button.

cProgressForm has a couple of behaviors I'd like to see different. First, you have to manually call the Show method to display it. I'd rather if it automatically displayed itself if necessary the first time SetProgress is called. Seconds, it sets a custom nSeconds property to SECONDS() in Init. This property can be used to determine the total time required for a process, but setting it in Init may be too soon. So, I created a subclass called SFProgressForm in SFXSource.VCX. The Init method uses DODEFAULT() and then sets nSeconds to 0. The SetProgress method simply sets Visible to .T. if the form isn't visible and nSeconds to SECONDS() if it was 0, and then uses DODEFAULT() for the usual behavior.

TestProgress.PRG is an example that searches the files in the FFC subdirectory of the VFP home directory for localized strings (constants with "_LOC" in the name). It uses SetMax to specify the total number of files to process, and updates the progress meter after each file has been searched. Uncomment the code setting lShowCancel or lShowThermometer to .F. to see what the progress dialog looks like in those cases.

```
* Figure out how many files we want to process.

lnFiles = adir(laFiles, home() + 'ffc\*.*')

* Instantiate the progress dialog and set some properties.

loProgress = newobject('cProgressForm', 'SFXSource.vcx')
loProgress.SetDescription('Looking for localized strings')
loProgress.SetMax(lnFiles)
*loProgress.lShowCancel      = .F.
*loProgress.lShowThermometer = .F.

* Search the files for localized strings, updating the progress meter as we go.

lnFound = 0
for lnI = 1 to lnFiles
  lcFile     = laFiles[lnI, 1]
  lcFullFile = forcepath(lcFile, home() + 'ffc')
  lcExt      = justext(lcFile)
  do case
    case inlist(lcExt, 'H', 'PRG')
```

```
      lcContents = filetostr(lcFullFile)
      lnFound    = lnFound + occurs('_LOC', upper(lcContents))
    case lcExt = 'VCX'
      use (lcFullFile) again shared
      lnFound = lnFound + occurs('_LOC', upper(METHODS))
  endcase
  if not loProgress.SetProgress(lnI, 'Checking ' + lcFile + '...')
    exit
  endif not loProgress.SetProgress(lnI, 'Checking ' + lcFile + '...')
next lnI

* Display the results.

messagebox(transform(lnFound) + ' instances found in ' + ;
  transform(seconds() - loProgress.nSeconds) + ' seconds.')
```

Note that cProgressForm uses the Microsoft ProgressBar ActiveX Control, so you'll need to distribute MSCOMCTL.OCX with your application. It also uses an animated GIF file, FindComp.GIF in the BitMaps subdirectory of the Toolbox XSource directory, so that file will be added to your project when you build it.

Outlook bar

Even since its release several years ago, Outlook has been influencing the appearance and behavior of other applications. One of the most commonly copied interface elements is the Outlook bar, a set of controls at the left edge of Outlook. These controls provide a collapsible set of categories, each of which contains individual items you can select that determine what appears on the right side of Outlook.

There are a number of ActiveX controls you can purchase that mimic the look of the Outlook bar, some of which work with VFP and many of which do not. However, there's an application that comes with VFP that uses a similar type of control, and we get the source code for it in XSource: the Toolbox.

The Toolbox was added in VFP 8 as a way of providing an interface similar to Visual Studio.NET for selecting controls and other frequently used items. Because it doesn't look exactly like Outlook, I'll call the control that maintains the categories and items under each one the "toolbox."

As you can see in **Figure 4**, the toolbox consists of four areas of interest (not counting the help text at the bottom). A category represents a set of items. Categories appear as bars in the toolbox. Only one category is expanded at a time; the rest appear as just the category bar. Click on the bar to expand a category; this causes the previously expanded category to close. A tool is an item within a category. Tools have an image, a name, and a tooltip. You can perform various actions on tools: clicking, right-clicking, dragging, and so forth. Also within a category are scroll up and down buttons. You don't have to click on these buttons; simply hover the mouse pointer over one for a second or so to start scrolling in the appropriate direction.

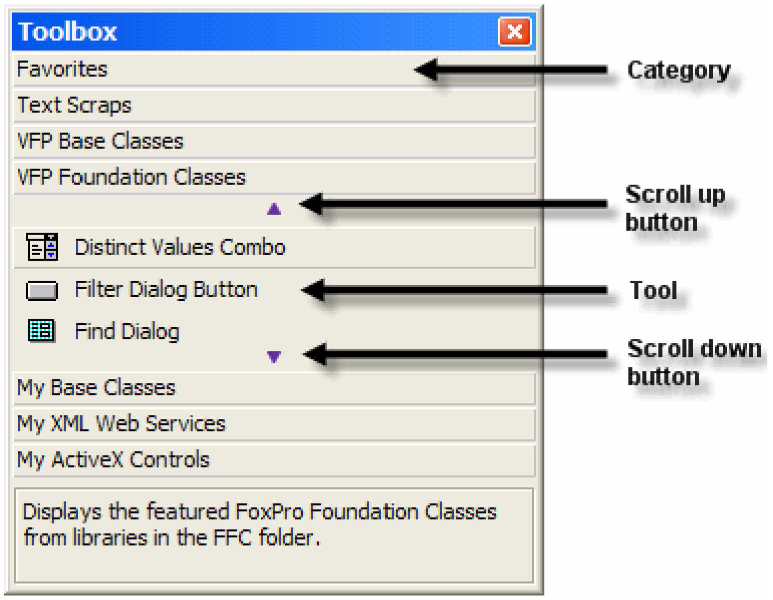


Figure 4. The components of the toolbox.

The classes that provide the toolbox are in FoxToolbox.VCX and _Toolbox.VCX in the Toolbox XSource directory. The class of interest in FoxToolbox.VCX is ToolContainer. It contains the controls and logic for a single category. A form that displays a toolbox will contain as many ToolContainer objects as there are categories to display. The oToolItem property of each ToolContainer has a reference to an _Category object (or one of its subclasses; different subclasses are used for different types of categories in the VFP Toolbox application) from _Toolbox.VCX. _Category has properties and methods for the category. For our use, _Category isn't really important, but some ToolContainer methods require that it exists.

Each ToolContainer also maintains a collection of tools, each of which is represented by an _Tool object (or one of its subclasses; different subclasses are used for different types of tools in the VFP Toolbox application) from _Toolbox.VCX. _Tool has properties and methods for a tool.

The main form in the Toolbox application is based on the ToolboxForm class in FoxToolbox.VCX. I originally considered subclassing this form, but unfortunately it's heavily dependent on performing tasks related to the Toolbox application rather than an generic toolbox. So, instead, I created SFToolboxForm (in SFXSource.VCX) as the basis for forms with a toolbox.

The ShowToolbox method, called from Init, is responsible for creating the toolbox.

```
local loCategories, ;
  lnI, ;
  loCategory, ;
  lcCategory, ;
  loContainer, ;
  loTools, ;
  lnJ, ;
  loTool
with This

* Get a collection of categories and process each one.

  loCategories = .GetCategories()
  .nCategories = loCategories.Count
  for lnI = 1 to .nCategories
    loCategory = loCategories.Item(lnI)

* Create a ToolContainer object and set its properties.

    lcCategory = 'Category' + loCategory.cID
    .NewObject(lcCategory, 'ToolContainer', 'FoxToolbox.vcx')
    loContainer = evaluate('.' + lcCategory)
    with loContainer
      .SetFont(This.FontName, This.FontSize, '')
      .oToolItem  = newobject('_Category', '_Toolbox.vcx')
      .CategoryID = loCategory.cID
      .Caption    = loCategory.cName
      .Width      = This.nToolboxWidth
      .Visible    = .T.
    endwith

* If this is the first category, set the standard height for categories to the
* height of this one and make it the default open one.

    if lnI = 1
      .nStandardHeight  = loContainer.Height
      .cCurrentCategory = loCategory.cID
    endif lnI = 1

* Get a collection of tools for the current category. For each one, bind its
* OnClick method to our ToolSelected method. The toolbox classes expect its
* oEngine property references an object with a DblClickToOpen property, so
* create one. Add the tool object to the category container.

    loTools = .GetToolsForCategory(loCategory.cID)
    for lnJ = 1 to loTools.Count
      loTool = loTools.Item(lnJ)
      bindevent(loTool, 'OnClick', This, 'ToolSelected')
      loTool.oEngine = createobject('Empty')
      addproperty(loTool.oEngine, 'DblClickToOpen', .F.)
      loContainer.AddTool(loTool)
    next lnJ
  next lnI

* Adjust the sizes of the categories.

  .ResizeToolbox()
endwith
```

This code starts by calling the GetCategories method to return a collection of objects with information about each category. GetCategories is abstract in this class because the exact mechanism to fill the collection will vary. The code then goes through the collection, adding a ToolContainer object to the form for each category. The width of the ToolContainer is set to the nToolBoxWidth property of the form, so be sure to set that property in a subclass or form created from this class.

For the first category only, the code sets the nStandardHeight property, which contains the standard (that is, closed) height for each category, to the height of the current container. It also sets cCurrentCategory to the ID for this category so it'll be opened automatically in code we'll see later.

Next, the code calls the GetToolsForCategory method, which is abstract in this class, to return a collection of _Tool objects representing the tools for the specified category. When a user clicks on a tool, the toolbox calls the OnClick method of the appropriate tool object. We want clicks to be handled at the form level, so the code uses BINDEVENT() to bind the OnClick method of each tool object to the ToolSelected method of the form. Because various toolbox methods expect that the oEngine property of a tool object references an object with a DblClickToOpen property, the code creates such an object. The tool object is then added to the collection of tools in the category container.

Finally, ShowToolbox calls the ResizeToolbox method to open the selected category (in this case, the first one) and close the others (we won't look at the code for that method; feel free to examine it yourself).

The SetCategory method is called from toolbox code when the user clicks on a category. All this code does is set cCurrentCategory to the ID for the selected category and calls ResizeToolbox to handle the new selection.

SFToolboxForm contains three other abstract methods: OnRightClick, called when the user right-clicks on a tool, ResetHelpFile, which sets the help file to the default one, and SetHelpText, which, in the Toolbox application, puts some information about the selected tool into the help area at the bottom of the form. These methods are necessary because some toolbox methods call them. Although they don't do anything in this class, you could implement the appropriate behavior in a subclass.

TestToolbox.SCX is a sample form based on SFToolboxForm. It uses a toolbox to show the different modules in an accounting application. It reads these modules from a MODULES table, which contains records for the categories (such as "Accounts Receivable" and "Order Entry") and the modules under each one (such as "Customers" and "Orders"). It has the following code in GetCategories:

```
local loCollection, ;
  loCategory
loCollection = createobject('Collection')
select NAME, ID ;
  from MODULES ;
  where PARENTID = 0 and ACTIVE ;
  into cursor _CATEGORIES
scan
  loCategory = createobject('Empty')
  addproperty(loCategory, 'cName', trim(NAME))
  addproperty(loCategory, 'cID',   transform(ID))
  loCollection.Add(loCategory)
endscan
use
return loCollection
```

This code fills a collection with objects having cName and cID properties from the categories in the MODULES table (those records with PARENTID = 0).

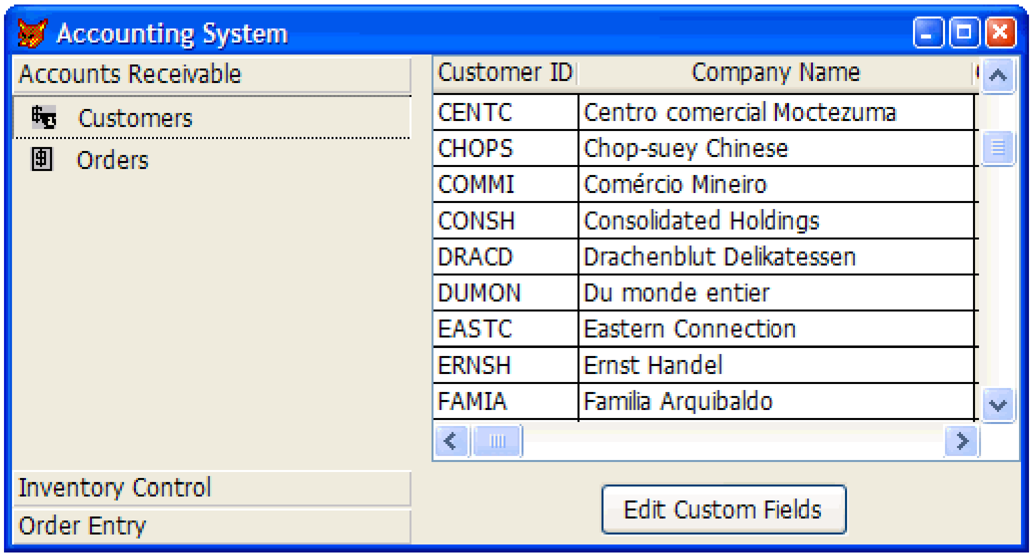
GetToolsForCategory is responsible for filling a collection of tools for the specified category.

```
lparameters tcID
local loCollection, ;
  loCategory
loCollection = createobject('Collection')
select NAME, ID, CLASS, LIBRARY, IMAGEFILE ;
  from MODULES ;
  where PARENTID = val(tcID) and ACTIVE ;
  into cursor _MODULES
scan
  loModule = newobject('_Tool', '_Toolbox.vcx')
  with loModule
    .ToolName  = trim(NAME)
    .UniqueID  = transform(ID)
    .ParentID  = tcID
    .ImageFile = textmerge(trim(IMAGEFILE))
    addproperty(loModule, 'cClass',   trim(CLASS))
    addproperty(loModule, 'cLibrary', trim(LIBRARY))
  endwith
  loCollection.Add(loModule)
endscan
use
return loCollection
```

This code selects the records from MODULES that have PARENTID containing the ID of the category to fill. Each object in the collection is an _Tool object, with its properties set appropriately. This code also adds a couple of custom properties: the class and library for a container of controls to display on the right side of the form when the tool is selected.

Thanks to BINDEVENT(), the ToolSelected method (we won't look at the code here) is called when the user clicks on a tool. ToolSelected adds an object to the right side of the form, using the cClass and cLibrary properties of the selected tool to indicate what class to use for the object.

Figure 5 shows what this form looks like when it runs.



Customer ID	Company Name
CENTC	Centro comercial Moctezuma
CHOPS	Chop-suey Chinese
COMMI	Comércio Mineiro
CONSH	Consolidated Holdings
DRACD	Drachenblut Delikatessen
DUMON	Du monde entier
EASTC	Eastern Connection
ERNSH	Ernst Handel
FAMIA	Familia Arquibaldo

Figure 5. The TestToolbox form shows how to create a form with a toolbox like the VFP Toolbox application or Outlook.

Creating a scrolling region

You may have come across a need to show controls in a scrolling region. For example, suppose you allow your users to add their own custom fields to your application. This is a great feature because it allows your users to customize the application to suit their needs. The problem is then how to display those custom fields. If there are only a few of them, it isn't a big deal, since they all fit on a form. If you have a lot, you could use a scrolling form. However, what if you need to show the fields on a form with other, non-scrolling controls? In that case, you need a scrolling region with the custom fields inside it.

The Task List is a relatively little-used application that's shipped with VFP since VFP 7. It allows you to add custom fields and displays them in a scrolling region on the Field page of the Task Properties dialog (see **Figure 6**).

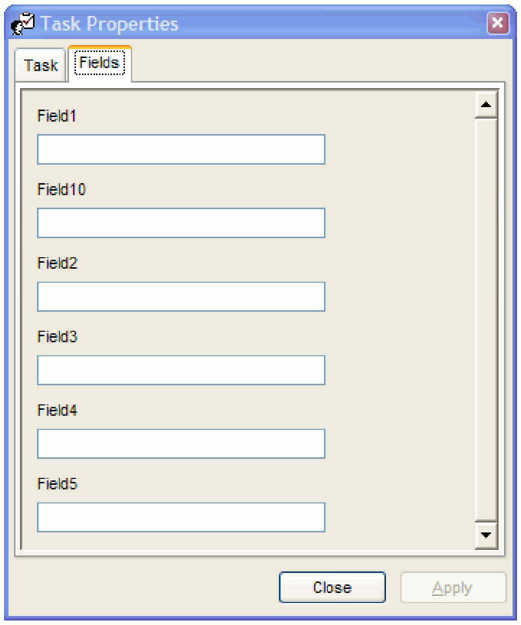


Figure 6. The Fields page of the Task Properties dialog in the Task List contains a scrolling region.

The scrolling region is an instance of the cntScrollRegion class in TaskListUI.VCX in the TaskList XSource directory. cntScrollRegion consists of a container (where the controls will go) and a Microsoft Flat ScrollBar

Control (you need to need to distribute MSCOMCT2.OCX with your application to provide this ActiveX control). Each of the controls added to the container (for example, the labels and textboxes in Figure 6) is an instance of a *Scrolling class (such as edtScrolling), also from TaskListUI.VCX. The reason special classes are required rather than base classes is so when the user moves from one control to the next, the scrolling region is automatically scrolled if necessary.

Let’s see how to use these classes in our own application. I decided to create custom fields for the Customers table in the sample Northwind database. Rather than adding fields to Customers directly, which causes issues when I want to update the structure of the table when I install an updated version of my application, I created a separate table, CustomFields, that has a CustomerID field that matches the CustomerID value of a record in Customers, plus whatever custom fields my users add. Another table, CustFldDef, defines the custom fields the user has added so far (presumably, I’d provide a dialog in which the user can define new fields and which updates the contents of CustFldDef as necessary). This table has the fields FIELDNAME, FIELDTYPE, FIELDLEN, CAPTION, ORDER, and PICTURE, which define information about each custom field.

SFCustomFieldsForm, in SFXSource.VCX, is a base class form with a cntScrollRegion on it and OK and Cancel buttons (the OK button is named cmdApply because some methods in the Task List classes require this). The Init method opens the custom fields table (the name of the table, the index order to use, and the name of the key field are contained in the cCustomFieldsTable, cCustomFieldsOrder, and cCustomFieldsKey properties rather than hard-coded to make this form generic) and calls the SetupCustomFields method to set up the scrolling region.

Here’s the code for SetupCustomFields:

```
local lnSelect, ;
  lcAlias, ;
  lnTop, ;
  lcField, ;
  lcType, ;
  lnLen, ;
  lcCaption, ;
  lcPicture, ;
  lcClass, ;
  lcTemplate, ;
  lcLabel, ;
  loLabel, ;
  loField
with This.cntScrollRegion

* The TaskList scrolling controls require that the form has a Task member with
* properties matching the field names. So, let's create it.

  This.AddProperty('Task', createobject('Empty'))

* Open the table containing information about the custom fields and go through
* each record.

  lnSelect = select()
  select 0
  use (This.cMetaDataTable) again shared order ORDER
  lcAlias = alias()
  lnTop   = 10
  scan

* Get information about the field.

    lcField    = trim(FIELDNAME)
    lcType     = FIELDTYPE
    lnLen      = FIELDLEN
    lcCaption  = trim(CAPTION)
    lcPicture  = trim(PICTURE)
    lcTemplate = ''

* Figure out what class to use for data entry based on the data type. For Date
* and DateTime fields, you could use the oleDateScrolling and oleTimeScrolling
* classes, except they can't handle blank dates.

    do case
      case lcType = 'L'
        lcClass = 'chkScrolling'
      case lcType = 'M'
        lcClass = 'edtScrolling'
      case lcType $ 'NFIBY'
        lcClass    = 'txtScrolling'
        lcTemplate = lcPicture
      otherwise
        lcClass    = 'txtScrolling'
        lcTemplate = replicate('N', lnLen)
    endcase

* If we aren't using a checkbox, create a label.

    if lcClass <> 'chkScrolling'
      lcLabel = 'lbl' + lcField
      .cntPane.NewObject(lcLabel, 'labScrolling', ;
        home() + 'Tools\XSource\VFPSource\TaskList\TaskListUI.vcx')
      loLabel = evaluate('.cntPane.' + lcLabel)
      with loLabel
        .Top      = lnTop
        .Left     = 10
        .Caption  = lcCaption
        .FontName = This.FontName
        .FontSize = This.FontSize
        .Visible  = .T.
        lnTop     = lnTop + .Height + 4
      endwith
    endif lcClass <> 'chkScrolling'

* Create a control to use for data entry for this field.

    .cntPane.NewObject(lcField, lcClass, ;
      home() + 'Tools\XSource\VFPSource\TaskList\TaskListUI.vcx')
    loField = evaluate('.cntPane.' + lcField)
    with loField
      if lcClass = 'chkScrolling'
        .Caption = lcCaption
      endif lcClass = 'chkScrolling'
      .Top           = lnTop
      .Left          = 10
      .Visible       = .T.
      lnTop          = lnTop + .Height + 10
      .ControlSource = This.cCustomFieldsAlias + '.' + lcField
      if inlist(lcClass, 'oleDateScrolling', 'oleTimeScrolling')
        .Font.Name = This.FontName
        .Font.Size = This.FontSize
      else
        .FontName = This.FontName
        .FontSize = This.FontSize
      endif inlist(lcClass ...
      if not empty(lcTemplate)
        .Width = txtwidth(lcTemplate, This.FontName, This.FontSize) * ;
          fontmetric(6, This.FontName, This.FontSize) + 12
      endif not empty(lcTemplate)
      if lcClass = 'txtScrolling'
        .InputMask = lcPicture
      endif lcClass = 'txtScrolling'
    endwith

* Add a property matching the field name to the Task member.

    addproperty(This.Task, lcField)
  endscan

* Refresh the scrolling region.

  .cntPane.Height = lnTop
  .SetViewPort()

* Clean up.

  use
  select (lnSelect)
endwith
```

It starts by adding a new property to the form called Task and putting an Empty object into it. The Task List classes require that the form's Task property contains an object with properties matching the name of the custom fields (it uses this for data binding rather than the fields in a table), so later we'll add properties to the Empty object to match those names. The code opens the table containing meta data for the custom fields, the name of which is stored in cMetaDataTable rather than hard-coding it. SetupCustomFields figures out what *Scrolling class to use as the control for the field based on its data type, and if the control isn't a checkbox, adds a label to the scrolling region (using the lblScrolling class). It then adds the appropriate control for the field and sets various properties, including ControlSource, which binds the control to the field in the custom fields table. A property with the same name as the field is added to the Empty object contained in the Task property. Finally, after all the controls have been added, the code sets the Height of the scrolling region as necessary (possibly taller than the form if there are lots of controls) and calls the SetViewPort method of the cntScrollRegion object, which adjusts the size, position, and other attributes of the scroll bar as appropriate.

There's a bug in the SetViewPort method of cntScrollRegion that prevents the scroll bar from appearing properly: comment out the following line of code:

```
This.oleScrollBar.LargeChange = This.Height - 20
```

TestScrolling.SCX is a form based on SFCustomFieldsForm. Its cCustomFieldsTable, cCustomFieldOrder, cCustomFieldsKey, and cMetaDataTable are set to the appropriate values to use the CustomFields and CustFldDef tables. You can either run this form and pass it a CustomerID value from the Customers table (such as "ALFKI") or run the TestToolbox form discussed in the last section, select the Customers module, select a customer in the list, and click on the Edit Custom Fields button. **Figure 7** shows how the TestScrolling form appears when it's been scrolled part way down.

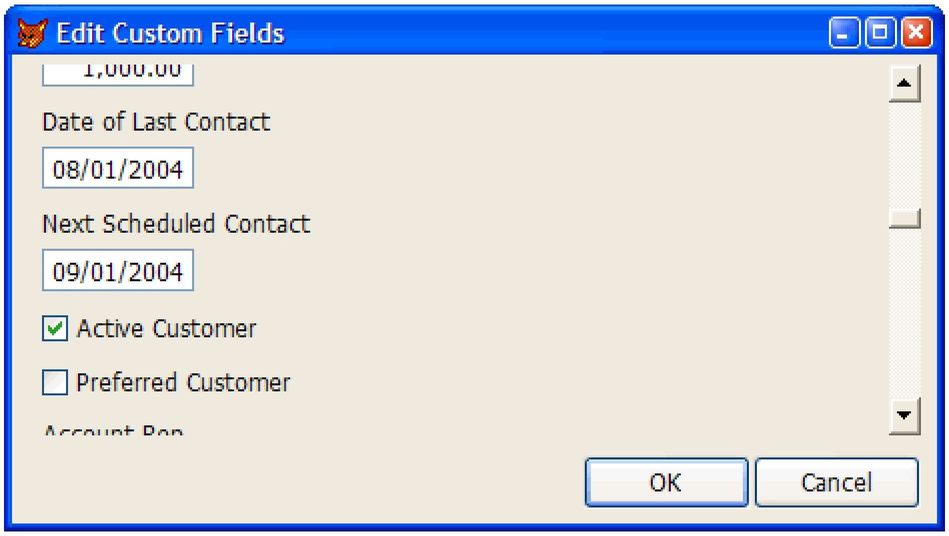


Figure 7. TestScrolling.SCX shows an example of how to use cntScrollRegion class.

Creating your own builders

Builder technology has been with VFP since its release. However, builders are fairly underutilized, partly because many people think they're difficult to create. While that's not really true, there are a lot of things to manage when creating a builder, such as getting a reference to the object being maintained by the builder, dealing with modality issues (if the builder is modeless, it should close when the form or class is closed), and so forth. Fortunately, XSource once again comes to the rescue.

XSource comes with a complete framework for creating builders. It isn't in its own directory, but is instead part of the CursorAdapter and DataEnvironment builders (in the DEBuilder subdirectory of the XSource Wizards directory). This framework makes it easy to create your own builders, because it handles all of the following issues:

- It can be called from Builder.APP (either as a registered builder in the Builder table in the Wizards subdirectory of the VFP home directory or as specified in a custom Builder property for an object) or as a standalone form.
- It supports both modal and modeless dialogs.
- It handles all of the builder plumbing issues, such as maintaining a reference to the object being maintained.
- It auto-refreshes on activate, so switching to the Properties window, making changes, and reactivating the builder updates the builder automatically.
- It automatically closes if the form or class is closed.
- It handles reverting changes by clicking on the Cancel button.

I won't discuss how builders work or builder architecture here, but the nice thing about the builder framework is that you don't have to understand these things to create your own useful builders. The only details you have to know are the following:

- To create a builder, subclass the BuilderForm class in BuilderControls.VCX. The controls you use in this form can be instances of the other classes in BuilderControls.VCX (such as BuilderLabel or BuilderTextbox) but they don't have to be; you can use any controls you wish.
- The simplest way to specify which builder is used for a particular class is to add a custom Builder property to the class and fill in the name of the class to use as the builder and the class library it's contained in, using the format *Library,Class*.

Let's look at an example where a builder would be useful. SFFile is a class in SFCCtrls.VCX that provides a control where a user can either type a file name or click on a button to display a dialog from which he or she can select a file. This class consists of a label, which provides a prompt for the control, a textbox, which contains the file name, and an instance of an SFGetFile object, a class in SFButton.VCX that displays an open file dialog. To use SFFile, simply drop it on a form and set a few properties, such as cExtensions (a list of the extensions to allow in the open file dialog), cCaption (the caption for the open file dialog), and cControlSource (something to bind the control to if desired). However, there are a couple of hassles:

- To set the Caption of the label, you have to right-click on the SFFile object, choose Edit, then click on the label, bring up the Properties window, select the Caption property, and type a new caption. Of course, you'll then have to move the textbox and SFGetFile buttons as necessary to make room for the label, and then likely have to resize the SFFile object if it's too small.
- If you want to make the SFFile object larger or smaller, you first have to resize it, then drill down into it and resize the textbox and move the SFGetFile button to the appropriate position.

Neither of these is a killer job, but they can take a moment or so. All of those little "moment or so" tasks in your day add up. Anything we can do to improve our productivity, especially for grunt-work tasks like this, is welcome.

Here are the steps to create a builder for SFFile:

- Since this class already has a custom Builder property, you don't have to add one. However, fill it in with the name of the class to use as the builder and the class library it's contained in, using the format *Library,Class*.
- Create a subclass of BuilderForm using the class and class library names you specified in SFFile.Builder. Set Caption to "SFFile Builder."
- Add a BuilderLabel object and set its Caption to "\<Label caption."
- Add a BuilderTextbox object beside the label, set its ControlSource to "Thisform.oSource.lblFile.Caption" and put the following code into its Validation method. This method is automatically called when the user moves off the control, and this code automatically adjusts the Left and Width properties of the textbox in SFFile as necessary to fit the caption for the label:

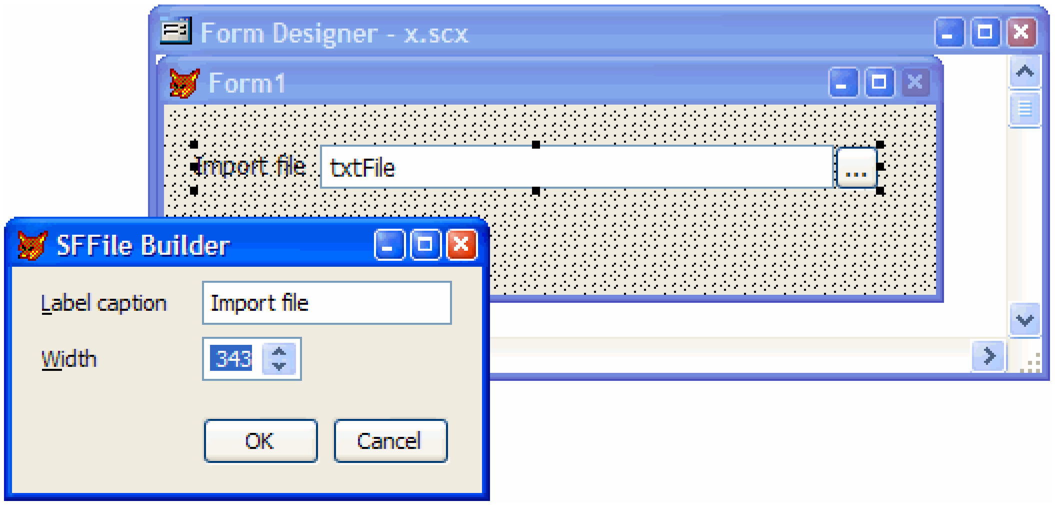
```
with Thisform.oSource
  .txtFile.Left  = .lblFile.Width + 5
  .txtFile.Width = .cmdGetFile.Left - .txtFile.Left
endwith
```

- Add another BuilderLabel and set its Caption to "\<Width."
- Add a BuilderSpinner object beside the new label, set its ControlSource to "Thisform.oSource.Width," and put the following code into its Validation method (this automatically adjusts the position of the button and the Width of the textbox in SFFile to fit the new width of the control):

```
with Thisform.oSource
  .cmdGetFile.Left = .Width - .cmdGetFile.Width
  .txtFile.Width   = .cmdGetFile.Left - .txtFile.Left
endwith
```

- Save and close the builder class.

To test this new builder, drop an SFFile object on a form, right-click, and choose Builder. Change the Label Caption and Width settings and watch the SFFile object adjust accordingly. **Figure 8** shows what this builder looks like in action.



Label caption: Import file
Width: 343

Figure 8. This builder took just a couple of minutes to create using the XSource builder framework.

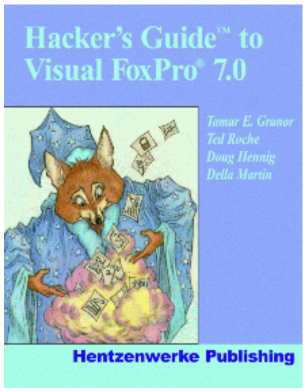
Note that the builder framework has changed somewhat since VFP 8 was released, so an updated version (which comes with VFP 9) is included with the source code accompanying this document.

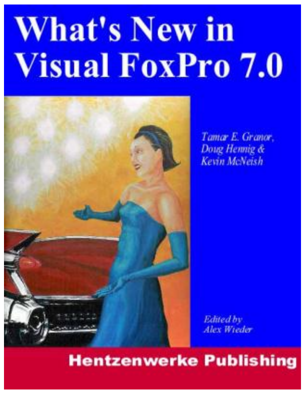
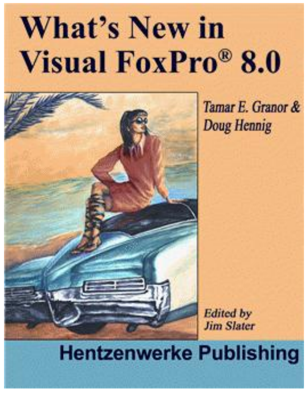
Summary

That many of the tools that come with VFP were written in VFP itself is interesting, but what's cool is the fact that we get source code for them. This makes it possible to change these tools to suit your exact needs or even reuse some of the code in these tools in your own applications. In this document, I discussed some of the components in XSource that I think are useful, but I want to encourage you to do your own spelunking in the XSource directories to see what useful pieces or techniques you might find.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT), the award-winning Stonefield Query, and the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro. Doug is co-author of the "What's New in Visual FoxPro" series (the latest being "What's New in Nine") 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). Doug formerly wrote the monthly “Reusable Tools” column in FoxTalk. He has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over the world. He is one of the administrators for the VFPX VFP community extensions Web site (http://www.codeplex.com/Wiki/View/aspx?ProjectName=VFPX). He has been a Microsoft Most Valuable Professional (MVP) since 1996.




What's New in Visual FoxPro 7.0
Tamar E. Granor, Doug Hennig & Kevin McNeish
Edited by Alex Wieder
Hentzenwerke Publishing

What's New in Visual FoxPro® 8.0
Tamar E. Granor & Doug Hennig
Edited by Jim Slater
Hentzenwerke Publishing