



VFPX 2018 Edition

Doug Hennig

Stonefield Software Inc.

Email: dhennig@stonefield.com

Corporate Web sites: www.stonefieldquery.com

www.stonefieldsoftware.com

Personal Web site : www.DougHennig.com

Blog: DougHennig.BlogSpot.com

Twitter: [DougHennig](https://twitter.com/DougHennig)

It's been three years since Rick Schummer last presented one of his excellent "deep dive" sessions into VFPX, so it's time for an update. We'll start by looking at the new GitHub-based VFPX site, then dive into several exciting projects.

Introduction

Since its inception, VFPX has been an incredible resource for VFP developers. It provides free, production-quality code from some of the best VFP developers on the planet over its 98 separate projects (as of September 27, 2018). I personally use 21 of these projects in my applications and am looking forward to using some of the new projects added in the last couple of years as the need arises.

VFPX has gone through a lot of changes in the past couple of years. Early in 2017, Microsoft announced that it was shutting down CodePlex. Over the years, the administrators of VFPX—Rick Schummer, Craig Boyd, Jim Nelson, and I—had discussed moving VFPX somewhere else due to numerous shortcomings in CodePlex, but the sheer volume of work involved in doing that had always discouraged us. However, now we had no choice: we had to find a new home.

GitHub seemed like the logical choice since it's the most popular open source site in the world. So, starting in April 2017, we secured the VFPX name on GitHub (it was previously registered to someone else but hadn't been used in years), set it up as an organization, and started migrating projects. Some of the work was done by the original project managers and some was done by us (those projects where the project manager could not be reached, had moved on from VFP, or were too busy).

Another change is that during the migration process, we decided to split the XSource and Sedna projects into separate projects for each component, both to make them more manageable and to make them easier to download and contribute to. For example, only the Data Explorer and Upsizing Wizard components of Sedna have had any significant changes since Sedna was released but you had to download the entire Sedna package just to get those updated versions.

A final change is that Jim Nelson has retired as a VFPX administrator as he is close to retirement in general. We thank him for all that he has done for the VFP community, not only as a VFPX administrator but as the developer of some of the most useful and popular VFPX projects (Thor, PEM Editor, and Finder, updated versions of GoFish and Code References, and contributions to FoxCharts and other projects) and as a speaker at Southwest Fox.

Let's start our look at VFPX 2018 by checking out GitHub.

VFPX on GitHub

Although Git (the distributed version control system, or DVCS) and GitHub (a cloud-based site for Git repositories) are extremely popular with developers world-wide, they aren't as well-known in the VFP community for a couple of reasons:

- VFP developers in general were slow in adopting version control as a best practice. I myself am in that category, only adopting it in 2011 for .NET projects and in 2013

for VFP applications. (I really don't know why I resisted for so long, as I could not live without it now.)

- For a while, Mercurial (another DVCS) and Bitbucket (another site for repositories) were more popular with VFP developers, partly because well-known VFP speakers such as Rick Borup and Alan Stevens spoke about them at conferences and those of us who adopted them promoted them to others in our community. I personally only started using Git and GitHub when we started migrating VFPX to GitHub.

So, I'm going to assume you aren't very familiar with Git or GitHub. This isn't intended to be exhaustive documentation for either; there are many sources of information on both, including <https://guides.github.com/> and <https://help.github.com/>. I found the GitHub for Windows Users course at <https://mva.microsoft.com/en-US/training-courses/github-for-windows-users-16749> very helpful. Also, Scott Hanselman's blog post "The Squishy Side of Open Source" (<http://feeds.hanselman.com/~527860891/0/scotthanselman~The-Squishy-Side-of-Open-Source.aspx>) has some good ideas for those new to open source.

Let's start with GitHub, since you don't actually need Git to work with any VFPX project.

GitHub

VFPX is located at <https://github.com/vfpv>. However, there isn't much to see there: just a list of repositories in reverse chronological order by last update. A better URL is <https://vfpv.github.io/>, the GitHub Pages site for VFPX (GitHub Pages provide a web site for a project). An even better URL is <http://vfpv.org>, which always redirects to the current location of VFPX (formerly vfpv.codeplex.com, now vfpv.github.io).

Whichever URL you use, the VFPX home page (**Figure 1**) is the first thing you see on the new site.

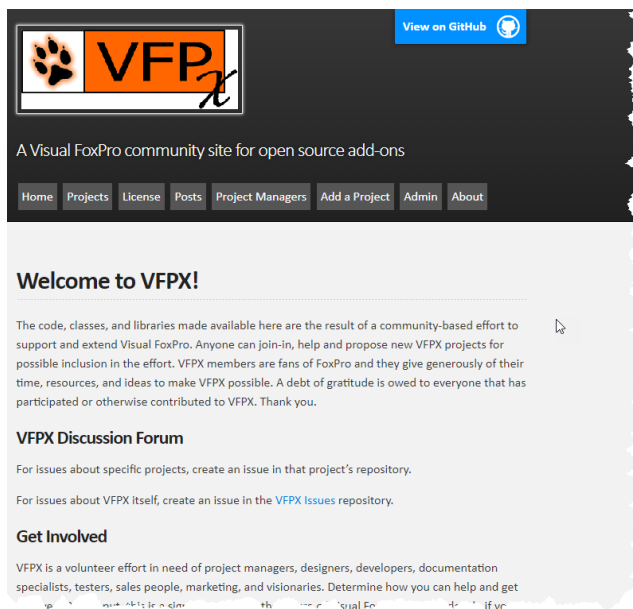
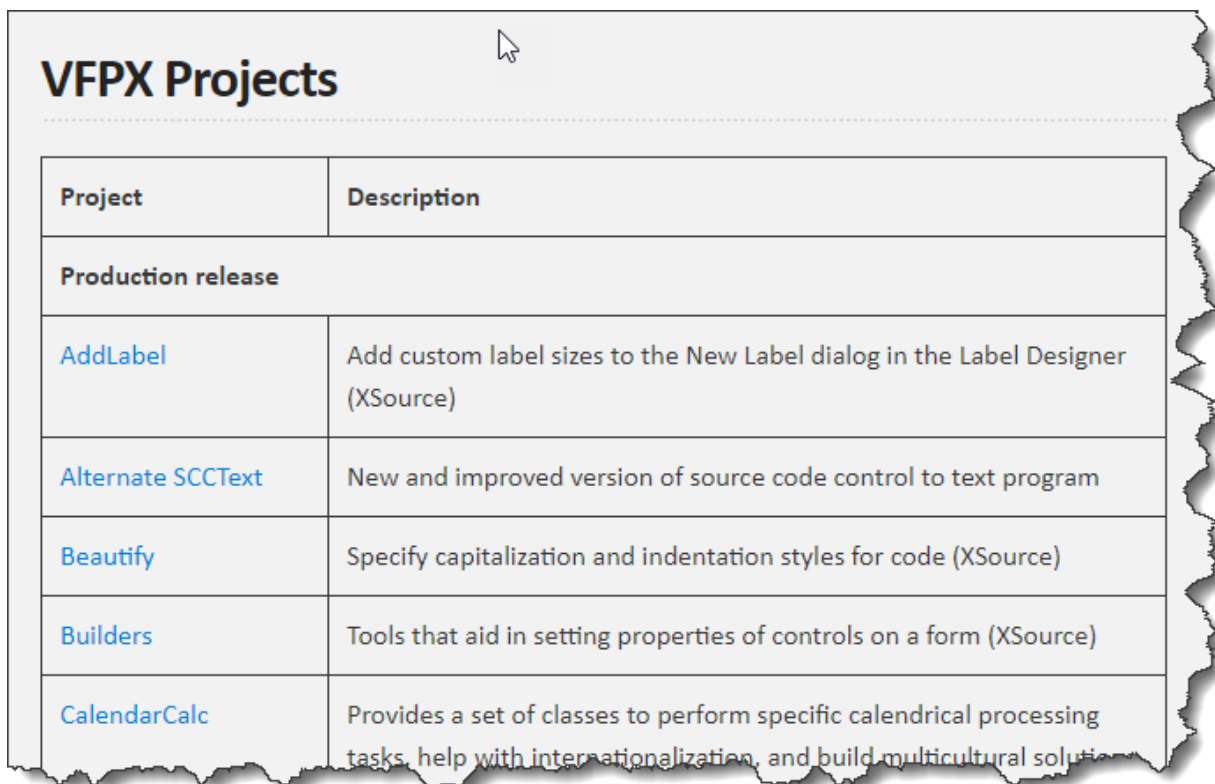


Figure 1. The VFPX home page.

The home page describes what VFPX is, how to get involved, and how to promote it. It also has a menu with the following items:

- Home: a link to the home page.
- Projects: a list of the projects including links to their repositories.
- License: the license all VFPX adhere to unless otherwise stated.
- Posts: news items about VFPX. There's an RSS feed for this page (the "subscribe via RSS" link at the bottom) so you'll be notified when posts are added.
- Project Managers: information for project managers.
- Add a Project: information about how to add a project to VFPX.
- Admin: information for the VFPX administrators (our own documentation).
- About: lists the VFPX administrators.

The Projects page (**Figure 2**) is the most commonly used page: it lists the projects by release type (Production, Release Candidate, Beta, Alpha, and Planning) and project name. The page also lists some other VFP open source projects that aren't considered to be part of VFPX. Click the name link to navigate to the repository for the project.



The screenshot shows the 'VFPX Projects' page. At the top, the title 'VFPX Projects' is displayed. Below the title is a table with two columns: 'Project' and 'Description'. The table is organized into sections, with the first section being 'Production release'. The projects listed are:

Project	Description
Production release	
AddLabel	Add custom label sizes to the New Label dialog in the Label Designer (XSource)
Alternate SCCText	New and improved version of source code control to text program
Beautify	Specify capitalization and indentation styles for code (XSource)
Builders	Tools that aid in setting properties of controls on a form (XSource)
CalendarCalc	Provides a set of classes to perform specific calendrical processing tasks, help with internationalization, and build multicultural solutions

Figure 2. The VFPX Projects page lists each project by name.



Some projects are hosted on GitHub under the VFPX organization (<https://github.com/vfpx/ProjectName>), some are under the author's repository (<https://github.com/Author/ProjectName>), and a few are on Bitbucket, as discussed later.

Figure 3 shows a typical VFPX project repository, in this case the one for the XLXSWorkbook project. As this document is not intended to be complete GitHub documentation, I won't go through everything, just the most common features.

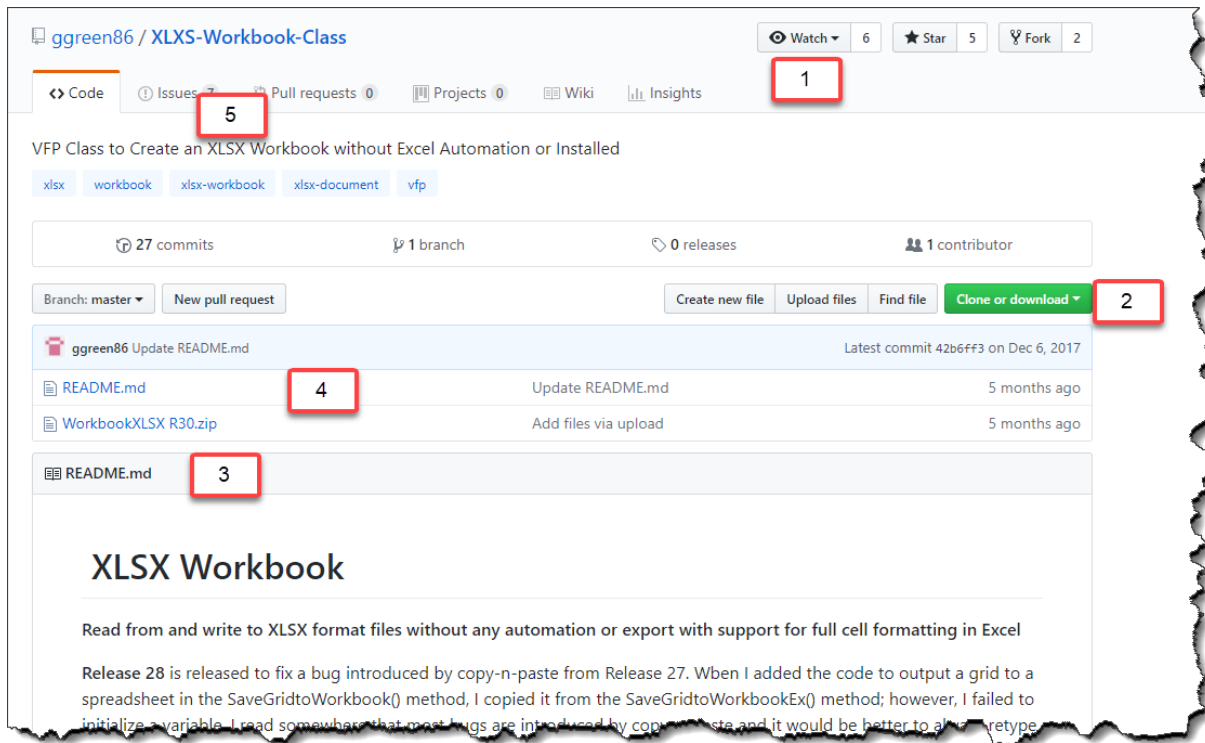


Figure 3. A typical VFPX project repository.

- You can get notification via email when the project is updated by clicking the Watch button (#1 in Figure 3) and choosing the desired option: Not Watching, Watching, or Ignoring. Note that you have to be logged in with a GitHub account to turn on watching.
- To install the project on your system, you can either download the project as a zip file or clone the repository, which requires have Git installed on your machine (#2 in Figure 3).
- The project's README.md file is a Markdown file (Markdown is similar to HTML but with simplified syntax) that GitHub automatically renders on the page (#3 in Figure 3) so it acts as the welcome page and in many case, provides the documentation for the project as well.
- The source list (#4 in Figure 3) shows the folder and files that make up the project. Some projects, such as this one, are just a zip file of source. Most, however, are

individual files, such as PRGs, SCXs, and so on. Click a file to view a page containing its content (most binary files can't be viewed, but some, such as PDFs can) and its version history.

- To report a bug, ask a question, or make a feature request, click the Issues tab (#5 in Figure 3) and create an issue. This is much preferred over sending an email to the project manager or posting something on a forum because it allows other users to see and comment on the issue as well.

Git

If you want to clone a project rather than just downloading it, you need to install Git. You can get Git from a variety of sources, including:

- The main source for Git is at <https://git-scm.com/downloads>. Choose the Windows version for VFPX projects.
- A lot of VFP developers use TortoiseGit (<https://tortoisegit.org/>), which provides a visual interface for Git (Git is basically a command-line utility). Installing TortoiseGit automatically installs Git as well.
- Some people like to use GitHub Desktop (<https://desktop.github.com/>), which is a visual interface for GitHub. Installing GitHub Desktop automatically installs Git as well.

To clone a repository, open a command window somewhere and type:

```
git clone RepositoryURL folder
```

where *RepositoryURL* is the URL displayed when you click the Clone or Download button in the repository and *folder* is the folder in which to clone the repository (the folder doesn't have to exist). If you use TortoiseGit, you can simply right-click some folder in File Explorer, choose Git Clone, and enter the URL and folder in the dialog that appears.

As I mentioned earlier, I'm not going to go into detail on how to use Git, as there are a lot of resources available to learn Git.

Thor Check for Updates

Another way to install a project is by installing Thor and then using its Check for Updates (CFU) function (**Figure 4**). Simply put a checkmark in the Update column for those projects you want to install or update and click Install Updates.

Thor CFU downloads files from a folder on a web site unrelated to GitHub. Since project managers don't have write access to that site, they have to submit update files to the VFPX administrators to put them in the proper folder. This mechanism is kind of clunky so it's been improved by adding support for updates to be downloaded directly from a project's GitHub repository. However, not many projects have been updated to do that yet, so the version number you see in the Thor CFU dialog doesn't necessarily match the one in the project's repository; the GitHub files could be newer than the ones in the VFPX site.

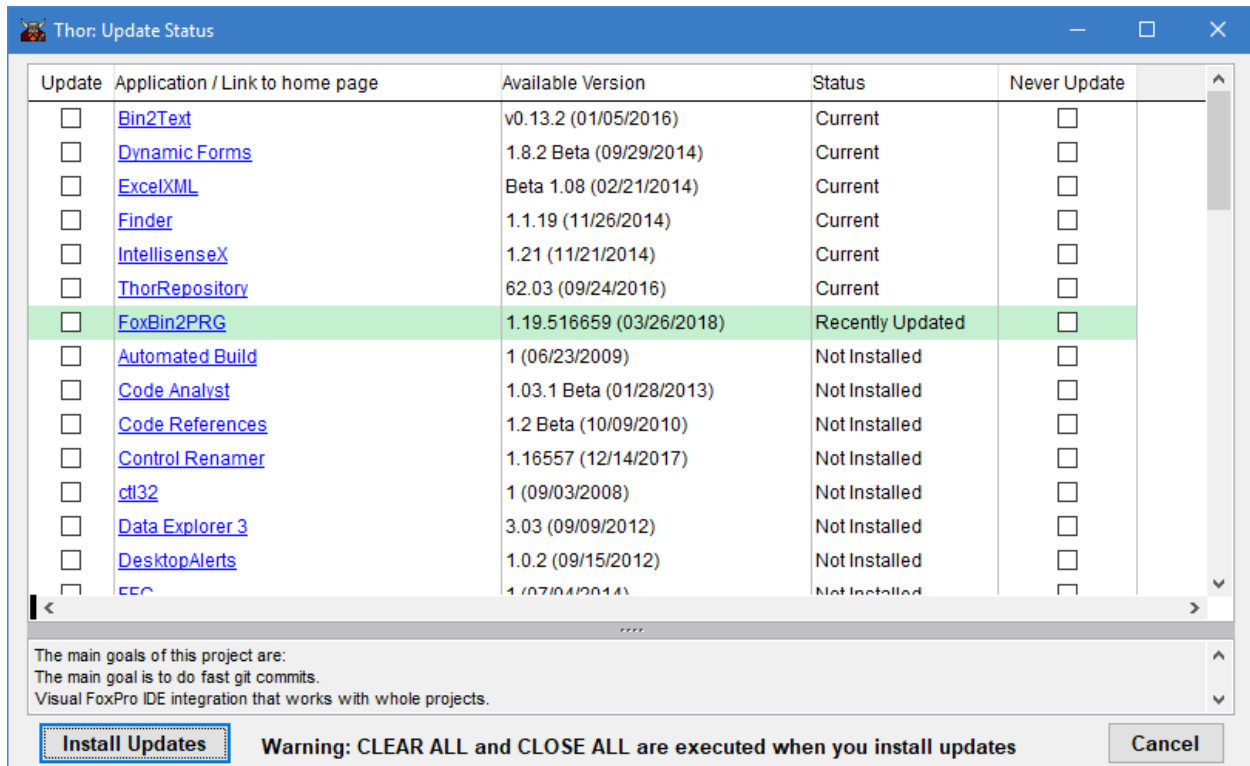


Figure 4. Thor's Check for Updates function can install and update many VFPX projects.

Now let's dig into what's new in VFPX.

Project Explorer

Project Explorer is a VFPX project (<https://github.com/DougHennig/ProjectExplorer>) that replaces the Project Manager with a modern interface and modern capabilities. It has most of the features of the Project Manager but adds integration with DVCS (including built-in support for FoxBin2PRG and optional auto-commit after changes), support for multiple projects within a "solution," allows you to organize your items by keyword or other criteria, and has support for easy "auto-registering" addins that can customize the appearance and behavior of the tool.

Thanks to many suggestions from the Fox community, I've made lots of improvements to Project Explorer since I presented it at Southwest Fox in October 2017 and the German DevCon in November 2017.

Creating new classes and forms

When you click the New button in the toolbar for classes, the New Class dialog (**Figure 5**) appears. It has similar functionality to that dialog in the Project Manager, with these additional features:

- *Based on* is set to the name of the selected class if there is one. This makes it easy to subclass an existing class by simply selecting it and clicking the New button.

- *From* is set to the selected VCX but it's a combobox containing the ten most recently used class libraries, so you can select one from the list. *Based on* adjusts to display the classes in the selected library. The libraries are listed in most recent to least recent order.
- You can create a new class by subclassing the *Based on* class or by copying it (the equivalent of dragging a class from one VCX to another and then renaming it in the Project Manager).

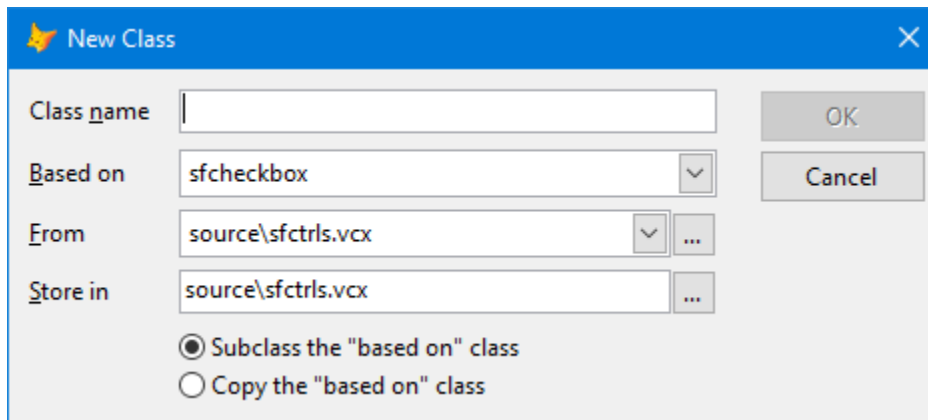


Figure 5. The New Class dialog displays when creating a new class.

There are a couple of ways to choose which class to base a new form on:

- When a form class is selected, right-click the class and choose Create Form from Class, then specify the name and path of the new form in the file dialog that appears.
- When forms are displayed, click the New button to display the New Form dialog (**Figure 6**). The *Based on* and *From* controls work the same as they do in the New Class dialog.

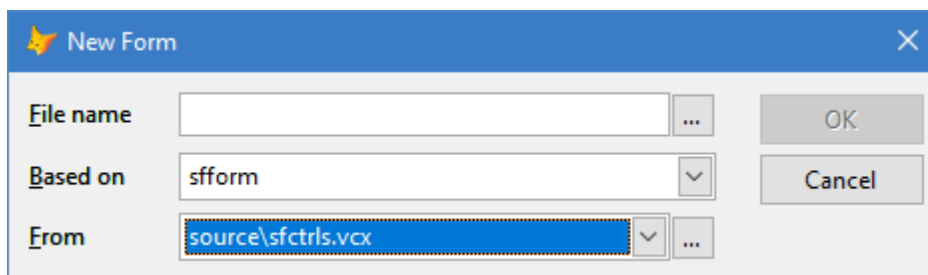


Figure 6. The New Form dialog appears when creating a new form.

Performance improvements

- At startup, the project was opened and closed multiple times in order to get the version control status of all files. That no longer happens.

- If only the text equivalents of binary files are stored in the repository, Project Explorer no longer closes the project when committing changes to a file or getting the version control status of a file.
- Rather than having a single TreeView control that's constantly emptied and reloaded as you change the selected tag, there's now one TreeView per tag. Selecting a tag the first time loads and displays that TreeView, including getting the version control status of every file in that tag, and hides the other ones; the next time that tag is selected, the TreeView isn't reloaded but is simply redisplayed. This makes the performance of switching tags in a large project much faster.

Version control improvements

- Project Explorer now works properly with Git even when TortoiseGit isn't installed.
- Support was added for other "binary to text" converters besides FoxBin2PRG; implementing them is left up to others.
- You can now change the setting of the *Binary Files in Repository* setting in the Version Control Properties dialog.
- The shortcut menu now has Convert Binary to Text and Convert Text to Binary functions. This is handy, for example, if you edit the records in a table for which the text equivalent is stored in the repository.
- Project Explorer now supports FoxBin2PRG configuration settings better.
- When version control is turned on for a solution, it now automatically detects if a repository folder exists and only prompts the user if one isn't found.
- When dragging a class to another VCX, Project Explorer assigns new UNIQUEID values to the members of the new class. This prevents an issue with duplicate values in FoxBin2PRG.

Other changes

- The *Class Library* and *Class Name* labels in the Servers tab of the project properties are now hyperlinked: clicking them takes you to the VCX or class, respectively.
- If Solution.xml exists in the current folder, it's opened automatically rather than prompting the user. Also, if there's only one PJX file in the current folder and no Solution.xml, Project Explorer automatically opens that project and creates a solution file for it.
- If you create a MyAddins subdirectory of the Addins folder in the directory where Project Explorer is installed, any addin PRGs in that subdirectory override addins in the Addins folder. This is handy because the addins that come with Project Explorer are disabled by default, so when a new version is released, the addins PRGs are overwritten and any you had enabled are disabled again. Simply copy the PRG into the MyAddins folder and enable it; it won't be overwritten when an updated version is installed.

- You can now define both the forecolor and backcolor for categories and specify which color is the *Category* combobox. Also, the solution is reloaded when you close the Category Editor dialog so changes are displayed immediately.
- You can now edit the OLEPublic and icon properties of a class in Project Explorer without having to use the Class Designer.
- The shortcut menu now has a Builder function which invokes the same builder or builder dialog you would see in the Project Manager.
- The shortcut menu for the *User* and *Description* editboxes now has a Zoom function that displays a resizable dialog in which you can view or edit the content of the editbox.
- Pressing Enter in the TreeView control now acts like double-clicking.
- If you manually close a project, Project Explorer automatically closes when it's activated.
- The Refresh button in the toolbar was replaced with a Sort/Filter button. Refresh is now available in the shortcut menu.
- Clicking the "..." button in the Sort and Filter and Assign Tags dialogs displays the VFP Expression Builder dialog.
- Project items now have a CategoryName property so you can filter on that rather than the category's ID number. Also, the Tags property is now a comma-delimited list of tags instead of a collection for easier filtering.
- There are a couple of new addins. AddWLCHackCXtoShortcutMenu.prg adds a Run HackCX Professional function to the shortcut menu to launch Hack CX Professional (<http://www.whitelightcomputing.com/prodhackcxpro.htm>). EditViewWithViewEditorPro.prg tells Project Explorer to edit a view using White Light Computing's ViewEditor Professional (<http://www.whitelightcomputing.com/provieweditorpro.htm>) rather than the VFP View Designer.
- The Run function now works for classes. For non-form classes, the class is instantiated and added to _SCREEN at position 0, 0. For form classes, the class is instantiated and a reference to it added to _SCREEN.
- Turn on the new *Add and New allow any file type* setting in the Options dialog to allow the Add and New functions to display a dialog in which you can choose any file type. Turn it off to only allow a file of the selected type to be chosen; for example, if a form is currently selected, Add and New only allow you to add or create a form.
- Turn on the *Remove unused headers* setting in the Options dialog to remove headers that don't have any items under them in the TreeView, such as "Labels" if there aren't any labels in the project.
- Project Explorer can now be installed or updated using the Thor Check for Updates function.

- Running Project Explorer with “do ProjectExplorer.app with '?'” displays a help message with the parameters you can pass to it.
- Numerous bugs were fixed.

Summary

I've been using Project Explorer in my day-to-day work for over a year now and find it a lot more useful than the VFP Project Manager. If you haven't tried it yet, take it for a spin. I look forward to your feedback.

Log4VFP

Log4NET is a well-respected diagnostic logging library for .NET applications. The Log4VFP VFPX project that I created, available at <https://github.com/VFPX/Log4VFP>, provides a wrapper for Log4NET so you can use it in VFP applications.

Include Log4VFP.prg in your project so it's built into the EXE. There are only a few files to deploy Log4VFP with your application:

- Log4VFP.dll, a C# DLL that provides the wrapper for Log4NET that can be used from VFP (the source code for it is in the Log4VFP subdirectory).
- Log4NET.dll, the Log4NET assembly.
- A configuration file that tells Log4NET how, when, and where to log information. There are several example config files that accompany Log4VFP.

Log4VFP requires Rick Strahl's wwDotNetBridge, so if you're not already using that in your application, you'll have to download it (<https://github.com/RickStrahl/wwDotnetBridge>), include wwDotNetBridge.prg in your project so it's built into the EXE, and deploy clrHost.dll and wwDotNetBridge.dll with your application.

Sample.prg (**Listing 1**) shows how easy it is to use Log4VFP in your VFP code. Start by instantiating the Log4VFP class in Log4VFP.prg, optionally set the cConfigurationFile and cUser properties to the name and path of the configuration file to use and the name of the current user, and call the Open method with the name and path of the file to log to. To write something to the log, call one of the Log* methods, such as LogInfo or LogError, depending on the level of the logging you want.

Listing 1. Sample.prg shows how easy it is to use Log4VFP.

```
local lcConfigFile, ;
    lcLogFile, ;
    lcUser, ;
    loLogger, ;
    loException as Exception

* Define some things.

lcConfigFile = fullpath('compact.config')
```

```
    && the name of the configuration file to use; use this for a compact log
*lcConfigFile = fullpath('verbose.config')
    && the name of the configuration file to use; use this for a verbose log
*lcConfigFile = fullpath('database.config')
    && the name of the configuration file to use; use this to log to a SQL
    && Server database
lcLogFile      = lower(fullpath('applog.txt'))
    && the name of the log file to write to
lcUser        = 'DHENNIG'
    && the name of the current user

* Initialize the logger.

loLogger = newobject('Log4VFP', 'Log4VFP.prg')
loLogger.cConfigurationFile = lcConfigFile
    && optional: uses a basic log4vfp.config (created if necessary) if not
    && specified
loLogger.cUser = lcUser
    && optional: uses Windows user name if not specified
loLogger.Open(lcLogFile)

* Log the application start.

loLogger.LogInfo('=====> App started at {0}', datetime())
loLogger.LogInfo('Application object created: version {0}', '1.0.1234')
loLogger.LogInfo('Using {0} build {1} {2}', os(1), os(5), os(7))

* Log that an error occurred.

try
    x = y
catch to loException
    loLogger.LogError('Error {0} occurred: {1}', loException.ErrorNo, ;
        loException.Message)
endtry

* Log a process.

wait window timeout 2 'Inserting a 2 second delay between logs (1)...'
loLogger.StartMilestone('=====> Started process')
wait window timeout 2 'Inserting a 2 second delay between logs (2)...'
loLogger.LogInfo('Process done')

* Shut down the logger and display the log.

release loLogger
modify file (lcLogFile) nowait
```

Summary

Log4VFP is very easy to use. Adding it to an existing application is a bit of work because you have to add method calls to write to the log file. However, you can start slowly, perhaps in the more error-prone sections of your application, and add more logging as time permits and needs arise.

Win32API

For many years, the go-to source of information on calling Windows 32 API (Win32API) functions from VFP was news2news.com, created and maintained by Anatoliy Mogylevets. The site included documentation on hundreds of functions, including sample VFP code. There was even an off-line viewer application (written in VFP, of course) available. Although most of the site was only accessible by subscription, there were many free pages as well.

Unfortunately, early in 2018, Anatoliy decided not to renew his site's hosting. Long-time VFP developer Tore Bleken, who had just finished spending many hours updating the Thor documentation for GitHub and was looking for a new challenge, asked Anatoliy for permission to move the content of news2news.com to VFPX and Anatoliy generously agreed. That means that the thousands of pages of documentation and samples not only live on, but are now completely free!

Tore took on the challenge of converting the former news2news content, much of which lives in VFP tables, into Markdown, the native formatting of GitHub and did a marvelous job as you can see at <https://github.com/VFPX/Win32API>. There are five main pages that form the starting point for the documentation:

- [functions_alphabetical.md](#), an alphabetical listing of the functions.
- [functions_group.md](#), a list of the functions by group.
- [Libraries.md](#), a list of the functions by library (DLL).
- [samples_alphabetical.md](#), an alphabetical listing of the samples, not by function name but by sample description.
- [samples_group.md](#), a list of the samples by group.

Figure 7 shows the functions_alphabetical page. An index at the top allows you to jump to the list of functions starting with a certain letter, and each function includes a brief description and a link to the page for that function. The functions_group page is similar but lists functions grouped by a “group” name; for example, the Clipboard group lists the 20 functions related to the Windows Clipboard.



List of functions in alphabetical order

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [Z](#) [_](#)

[accept](#)

The accept function permits an incoming connection attempt on a socket.

[ActivateKeyboardLayout](#)

The ActivateKeyboardLayout function sets the input locale identifier (formerly called the keyboard layout handle) for the calling thread or the current process. The input locale identifier specifies a locale as well as the physical layout of the keyboard.

[AddClipboardFormatListener](#)

Places the given window in the system-maintained clipboard format listener list.

Figure 7. The functions_alphabetical page lists Win32API functions in alphabetical order.

Clicking a function link takes you to the page for that function. The page typically shows the function name, what group it belongs to (along with a link to the group on the functions_group page), the library it's in (including a link to the library on the Libraries page), the description of the function, links to VFP code samples, the C declaration for the function and it's VFP equivalent, descriptions of any parameters and the return value, comments about the function, and links to any related functions. **Figure 8** shows an example of a function page: the documentation for the ShellAbout function. As you can see, not every function has sample code, especially the lesser-used functions like ShellAbout.

[Home](#)

Function name : ShellAbout

Group: [Shell Functions](#) - Library: [shell32](#)

Displays a Shell About dialog box

Declaration:

```
int ShellAbout (  
    HWND    hWnd,           // handle of parent window  
    LPCTSTR szApp,         // title bar and first line text  
    LPCTSTR szOtherStuff, // other dialog text  
    HICON   hIcon          // icon to display  
);
```

FoxPro declaration:

```
DECLARE INTEGER ShellAbout IN shell32;  
    INTEGER hWnd;  
    STRING  szApp;  
    STRING  szOtherStuff;  
    INTEGER hIcon
```

Parameters:

hWnd Identifies a parent window

szApp Points to text that the function displays in the title bar of the Shell About dialog box

szOtherStuff Points to text that the function displays in the dialog box after the version and copyright information

hIcon Identifies an icon that the function displays in the dialog box

Return value:

If the function succeeds in displaying the dialog box, the return value is TRUE

Figure 8. The ShellAbout page shows the documentation for the ShellAbout function.

Let's look at a simple example. Suppose we don't like the way the VFP FULLPATH() function always returns a path in upper case. Go to the samples page and search for "case;" there are several hits but "Converting path to original case" sounds like it does what we

want. Click the link to that sample and copy and run the sample code. Indeed, it does return a path using the correct case. Turning that into a reusable function is a simple matter once you have the working code.

If you want to view the documentation off-line, clone or download the repository and copy the APIViewer and Data folders somewhere. You can either run APIViewer.prg or create an EXE from APIViewer.pjx and run that. Either way, the APIViewer form shown in **Figure 9** displays, allowing you to search for functions, samples, and constants by keyword. The Constants tab is especially useful since it isn't available in the online documentation and even MSDN articles on the Win32API functions usually specify a constant name rather than its value.

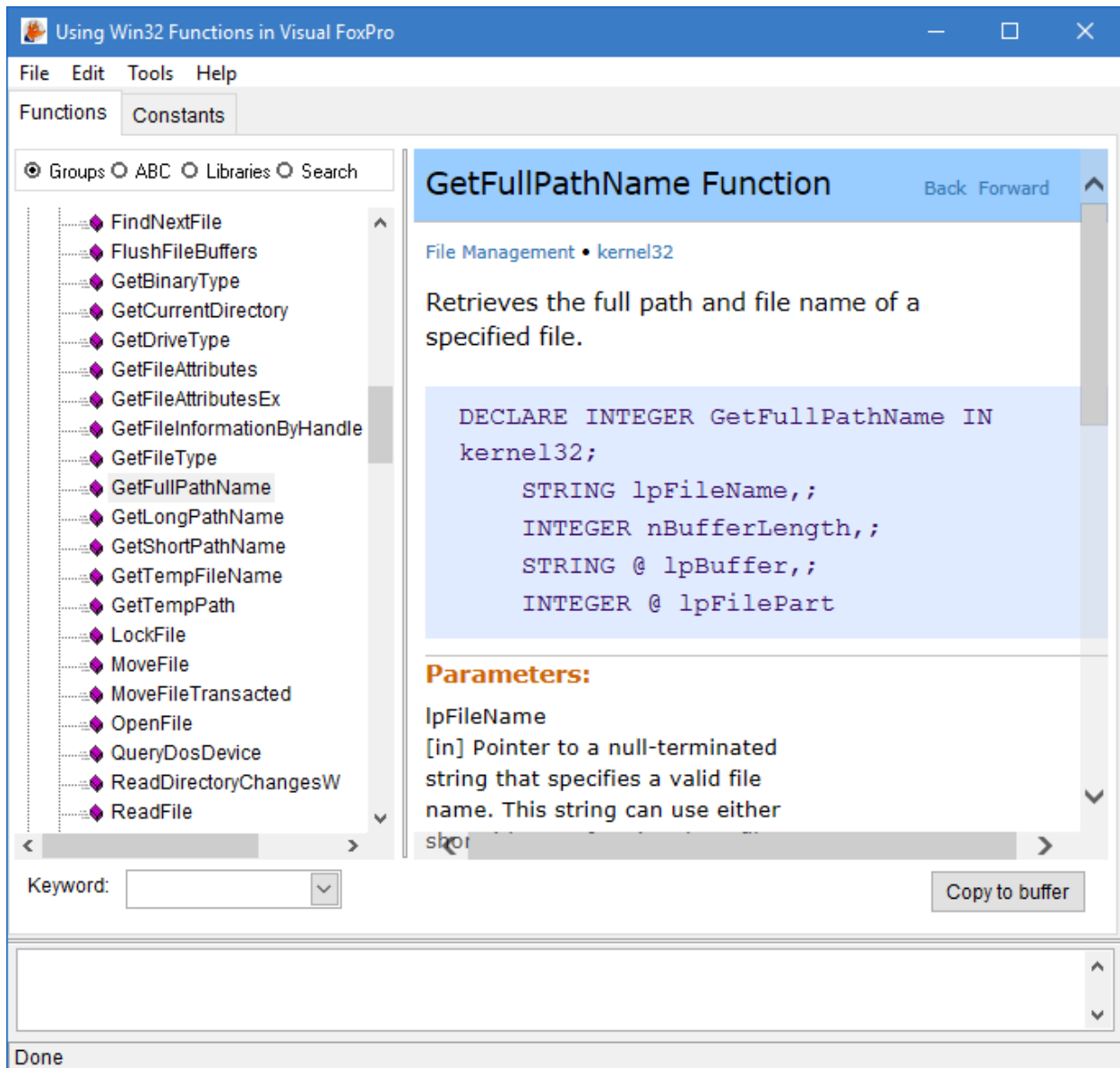


Figure 9. The APIViewer application allows you to view the Win32API documentation in a VFP form.

Summary

Win32API is a wonderful resource for VFP developers; thanks to Anatoliy for doing all the original work and kindly making it available to VFPX, and to Tore for the many hours he spent creating the Markdown pages on GitHub.

iCal4VFP

iCal4VFP is one of five new projects from António Lopes. Unlike most other VFPX projects, these projects are hosted on Bitbucket rather than GitHub because that's where António had them before contributing them to VFPX. Because they're on Bitbucket, downloading the source is a little different: click Downloads in the sidebar, then click "Download repository" (see **Figure 10**). You can also clone the repository by clicking the Clone button on the Source page to get the appropriate Git command.

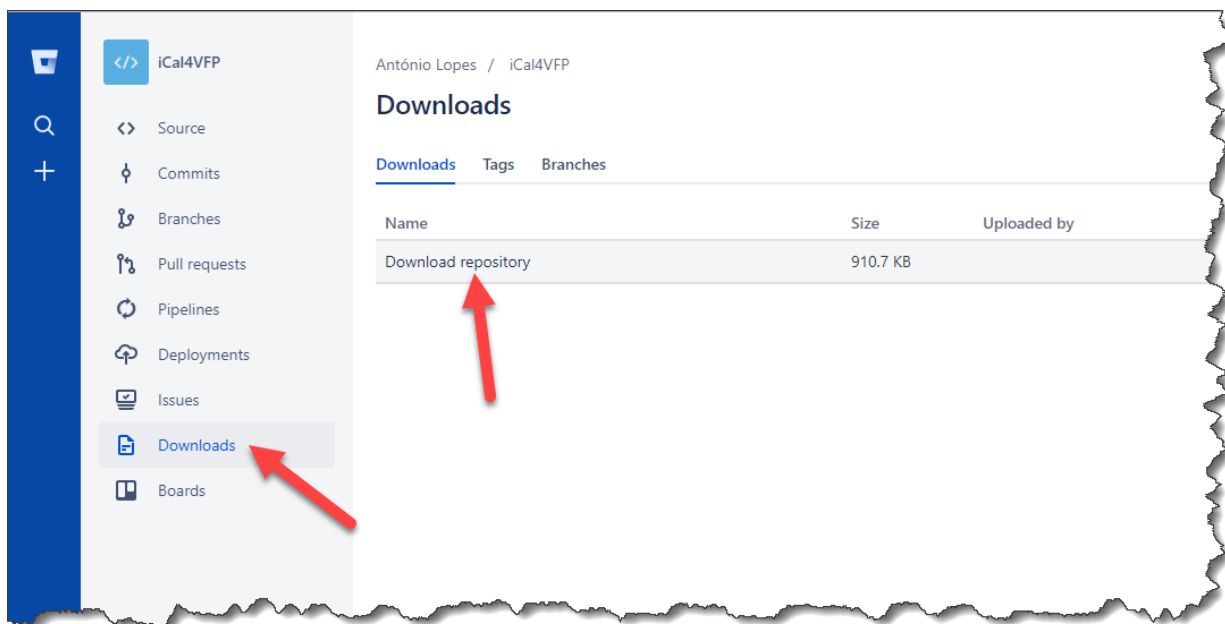


Figure 10. Downloading from Bitbucket is a little different than from GitHub.

iCal4VFP, available at <https://bitbucket.org/atlopes/ical4vfp>, is a library that helps VFP work with iCalendar files. According to Wikipedia (<https://en.wikipedia.org/wiki/ICalendar>), "iCalendar is a computer file format which allows Internet users to send meeting requests and tasks to other Internet users by sharing or sending files in this format through various methods. The files usually have an extension of .ics. With supporting software, such as an email reader or calendar application, recipients of an iCalendar data file can respond to the sender easily or counter-propose another meeting date/time. The file format is specified in a proposed internet standard (RFC 5545) for calendar data exchange." For example, although we haven't done it since 2013, the Schedule page of the Southwest Fox web site (<http://www.swfox.net/2013/schedule.aspx> for example; see **Figure 11**) used to include links to ICS files for each session, the idea being that you could click a link to add that session to your calendar.







Thursday, October 17				
9:00 -	 Donnay eXpress++ Workshop Part 1	 Fay Workshop on Migrating Applications to Lianja	 Giard HTML5 is the Future of the Web	
1:00 -	 Donnay eXpress++ Workshop Part 2	 Vinitsky VFP and HTML: Run Apps on Web with FoxInCloud	 MacNeill Moving Forward from FoxPro: Tools and Tech	

Figure 11. Clicking the link downloaded an ICS file to add the session to your calendar.

As noted in the Classes page in the repository, there are a lot of classes in this library. Fortunately, you only need to work with a few of them, as many are simply classes containing a few properties. For example, there's one class for each property in the iCalendar specification, such as iCalPropDURATION for the Duration property. The main classes you'll work with directly are iCalendar, defined in iCalendar.prg (all of the classes are defined in PRGs rather than VCXs), and ICSProcessor, defined in ICSProcessor.prg.



iCal4VFP requires another library, Tokenizer, which you can get from <https://bitbucket.org/atlopes/tokenizer>. It's just a single PRG which you can copy into the same folder as the iCal4VFP files.

Although it can do a lot of iCalendar-related tasks, we'll take a look at two: creating an ICS file so someone can add an event to a calendar, and processing an ICS file.

The code shown in **Listing 2**, taken from iCalendarTest.prg that accompanies this white paper, shows how to create an ICS file using iCal4VFP. iCalLoader.prg runs each of the iCal4VFP programs, which SET PROCEDURE ADDITIVE to themselves. iCalendar is the main class for working with iCalendar objects. Its AddICComponent method adds a new "component" to the object, in this case one representing a VEVENT. The AddICProperty method of the event object sets the values of the various properties of the event and the Serialize method generates the ICS file content as a string, which is then written to a file. Run this code to generate and display the file, then double-click it in File Explorer to add the event to your calendar.

Listing 2. iCalendarTest.prg shows how to create an ICS file using iCal4VFP.

```
* Get some constants we'll use.
```

```
#include iCal4VFP\Source\icalendar.h
```

```
* Set the path to find all iCal4VFP files and load the libraries.
```

```
set path to iCal4VFP\Source
```

```
do iCalLoader

* Create an iCalendar object.

loCalendar = createobject('iCalendar')

* Add a VEVENT component to the iCalendar object.

loEvent = loCalendar.AddICComponent(createobject('iCalCompVEVENT'))

* Add properties to the VEVENT component.

text to lcDescription noshow
It's been three years since Rick Schummer last presented one of his excellent
"deep dive" sessions into VFPX, so it's time for an update. We'll start by looking at
the new GitHub-based VFPX site, then dive into several exciting projects.
endtext
loEvent.AddICProperty('DTSTAMP',    datetime(), ;
    ICAL_DATE_IS_NOT_UTC)
loEvent.AddICProperty('UID',        sys(2015))
loEvent.AddICProperty('DTSTART',    {^2018-10-19 14:00:00}, ;
    ICAL_DATE_IS_NOT_UTC)
loEvent.AddICProperty('DTEND',      {^2018-10-19 15:15:00}, ;
    ICAL_DATE_IS_NOT_UTC)
loEvent.AddICProperty('CATEGORIES', 'Southwest Fox')
loEvent.AddICProperty('LOCATION',     'Flagstaff')
loEvent.AddICProperty('SUMMARY',     'VFPX 2018 Edition')
loEvent.AddICProperty('DESCRIPTION', lcDescription)

* Save the ICS file.

strtofile(loCalendar.Serialize(), 'VFPX2018.ics')
modify file VFPX2018.ics nowait
```

ReadICSFile.prg (**Listing 3**) does just the opposite: it reads the content of an ICS file into a cursor. It uses the ReadFile method of the ICSPProcessor class to read in the file and return an iCalendar object, then calls the ICSToCursor method to create a cursor named Temp containing the events in the file.

Listing 3. ReadICSFile.prg reads an ICS file into a cursor.

```
* Set the path to find all iCal4VFP files and load the libraries.

set path to iCal4VFP\Source
do iCalLoader

* Read the ICS file into a cursor.

loProc      = createobject('ICSPProcessor')
loCalendar = loProc.ReadFile('VFPX2018.ics')
if not isnull(loCalendar)
    loProc.ICSToCursor(loCalendar, 'EVENTS', 'temp')
    browse nowait
endif not isnull(loCalendar)
```

The Examples folder that comes with iCal4VFP has several other examples, including one showing working with time zones.

Summary

iCal4VFP can be used for a lot of iCalendar tasks but likely the most common is creating ICS files to add an event to a calendar. This is useful if your application deals with appointments or other calendar-related items where automatically adding an event to the user's calendar would be a nice feature. Although you can generate an ICS file yourself in VFP using text merge functions, iCal4VFP makes it a little easier.

CalendarCalc

According to its description, the CalendarCalc project, also from António Lopes, “provides a set of classes to perform specific calendrical processing tasks, help with internationalization, and build multicultural solutions.” It's basically a library that assists with calendar calculations and date math but it also includes a pretty nice date picker control.

CalendarCalc is available at <https://bitbucket.org/atlopes/calendar>, including both source files and documentation.

As with iCal4VFP, all the classes that make up CalendarCalc are in PRGs except the date picker control. The base class is CalendarCalc, defined in Calendar.prg. Subclasses of it define specific calendars, such as BritishCalendar in British-Calendar.prg; see the Classes page of the repository for a list of the available calendar classes, which PRG they're defined in, what dependencies they have, and which XML file they use for locale information. The PEM page lists the properties and methods of the classes.

Let's look at some examples to see how CalendarCalc can be useful. The first is determining the number of days between two dates. “Easy,” you're probably thinking, “simply use Date2 – Date1.” Ah, but what if you want to know the number of business days between the dates? After all, it doesn't make sense to say that it took 3 days to process an order that was received on Friday and shipped on Monday if the company isn't open on weekends. To complicate things further, what if the item was ordered from an American company on Friday June 29, 2018 and shipped on Friday July 6, 2018? That isn't seven days; it's four days because of the weekend and the American Independence Day (July 4) holiday.

To handle a business day calendar, use the BusinessCalendar class. To register US holidays with it, attach a USCalendarEvents processor to it (several event processors can be attached in case you need to handle events in multiple countries). You can then use methods of the BusinessCalendar object to determine dates, such as the code shown in **Listing 4**.

Listing 4. This code shows how to create a business calendar with US holidays registered.

```
* Set the path to find all CalendarCalc files and load the libraries we need.
```

```
set path to CalendarCalc
do Business-Calendar
do British-Calendar
```

* Create a BusinessCalendar and attach US calendar events for 2018 to it.

```
loCalendar = createobject('BusinessCalendar')
loCalendar.AttachEventProcessor('usa', 'USCalendarEvents', ;
    'us-calendar-events.prg')
loCalendar.SetEvents(2018)
```

* See the calendar events.

```
loCalendar.EventsToCursor('Temp')
browse
use
```

* Set the current date as June 29, 2018 (which is a Friday).

```
loCalendar.SetDate(2018, 6, 29)
```

* Display how many days later is July 6, 2018 (which is the next Friday).

* Unfortunately, it doesn't take into account just business days.

```
lnDays = abs(loCalendar.DaysDifference(2018, 7, 6))
messagebox('An order placed on 2018-06-29 and shipped on 2018-07-06 took ' + ;
    transform(lnDays) + ' calendar days to process')
```

* Instead, we'll calculate it by advancing business days until we get to the
* shipped date.

```
lnDays = 0
do while loCalendar.ToSystem() < date(2018, 7, 6)
    loCalendar.AdvanceBusinessDays(1)
    lnDays = lnDays + 1
enddo while loCalendar.ToSystem() < date(2018, 7, 6)
messagebox('An order placed on 2018-06-29 and shipped on 2018-07-06 took ' + ;
    transform(lnDays) + ' business days to process')
```

As you can see from the code, it would be nice if the DaysDifference method took weekends and holidays into account; instead, it returns the number of calendar days difference between the reference date (the date stored in the object) and the specified date. So, the code uses a more brute force method: advancing the business date to the next day in a loop until we get to the desired date. So, while this isn't ideal, it's still a lot less code than you'd write to handle this yourself.

Where CalendarCalc really shines is handling different calendar systems, especially if you have to deal with historic dates. For example, France switched to the Gregorian calendar on December 20, 1582 but Germany didn't switch until January 11, 1583. CalendarCalc handle dates that fall between those two, making sure they align properly. A more modern example is differences between the Gregorian, Hebrew, and Islamic calendars. Seasons.prg,

one of the sample programs that comes with CalendarCalc, shows the starting date of each season in all three calendars. For example, for the start of spring:

Gregorian: 20, March, 2018

Hebrew: 4, Nisan, 5778

Islamic: 3, Rajab, 1439

CalendarCalc provides two date picker controls: `TextDatePicker`, for use in container and forms, and `ContDatePicker`, which was designed for use in grids. Both of these controls are in `DatePicker.vcx`. I didn't look at `ContDatePicker`, but using `TextDatePicker` is easy: drop it on a form, set the `IsDateTime` property to `.T.` to support datetime values or leave it at the default `.F.` for date only, and set `ControlSource` as you would any control. If you want it to support more advanced CalendarCalc behavior, such as calendar events, add code to the `SetCalendars` method to instantiate the desired CalendarCalc calendar class and add it to a collection stored in the `Calendars` property. For example, the following code, taken from the sample `DatePicker` form that comes with CalendarCalc, handles US holidays:

```
DO LOCFILE("us-calendar-events.prg")

This.Calendars = CREATEOBJECT("Collection")

LOCAL USCalendar AS GregorianCalendar

m.USCalendar = CREATEOBJECT("GregorianCalendar")
m.USCalendar.AttachEventProcessor("usa", "USCalendarEvents")

This.Calendars.Add(m.USCalendar)
* we won't use the system calendar
This.CalendarIndex = 1
This.NoSystemCalendar = .T.
```

Figure 12 shows what the calendar looks like when the user clicks the arrow beside the textbox. Calendar events are highlighted and have a tooltip with the name of the event. The “hamburger” menu provides functions such as displaying the months in a year or years in a decade or century. The arrow buttons move to the next or previous month. Selecting a date and clicking the checkmark button closes the calendar.

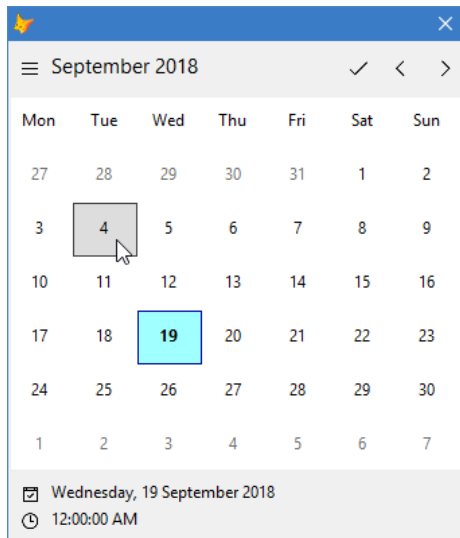


Figure 12. The date picker control is a visually appealing control and it supports calendar events.

Summary

CalendarCalc is a little obscure but could be handy if you need to deal with different calendars or a business calendar that takes holidays and other events into account. If you aren't already using a date or datetime picker, the ones that come with CalendarCalc are very nice.

Name Syntax Checker

The third project from António Lopes, Name Syntax Checker (<https://bitbucket.org/atlopes/names>), is quite specialized: it translates a name into one that's correct for a particular use. For example, there are certain naming rules for Windows file names: they cannot contain certain characters (<, >, :, ", /, \, |, ?, and *) and cannot match certain reserved words, such as "NUL" and "LPT1."

The Namer class is defined in Namer.prg and it works with specific processors, shown in **Table 1**.

Table 1. Processors used by Name Syntax Checker.

Class	Library	Purpose
JSONNamer	json-names.prg	JSON elements
MSSQLNamer	mssql-names.prg	SQL Server, using underscores
MSSQLDelimitedNamer	mssql-names.prg	SQL Server, using [and] as delimiters
MySQLNamer	mysql-names.prg	MySQL, using underscores
MySQLQuotedNamer	mysql-names.prg	MySQL, using ` as delimiters
VFPNamer	vfp-names.prg	VFP variable, table, and field names
VFPShortNamer	vfp-names.prg	VFP names limited to 10 characters (tags and fields in free tables)
WindowsNamer	windows-names.prg	Windows file and folder names

XMLNamer	xml-names.prg	XML element and attribute names
----------	---------------	---------------------------------

To create a valid name for a specific domain, instantiate Namer, attach the desired processor (you can attach multiple processors if desired), set the original name, and then call GetName.

Listing 5, taken from TestNamer.prg, demonstrates the use of Name Syntax Checker.

Listing 5. TestNamer.prg shows how to use Name Syntax Checker.

* Set the path to find all the files we need and load the libraries.

```
set path to NameSyntaxChecker
do Namer
```

* Create the controller and attach the processor for XML names.

```
loController = createobject('Namer')
loController.AttachProcessor('XMLNamer', 'xml-names.prg')
```

* Display the correct XML version of a name.

```
loController.SetOriginalName('#xml element name')
lcName = loController.GetName('XMLNamer')
messagebox('XML: ' + lcName)
```

* Attach the processors for SQL Server and MySQL names and display the correct
* versions of names containing a space and a reserved word.

```
loController.AttachProcessor('MSSQLDelimitedNamer', 'mssql-names.prg')
loController.AttachProcessor('MySQLQuotedNamer', 'mysql-names.prg')
loController.SetOriginalName('Company Name')
lcMSSQL = loController.GetName('MSSQLDelimitedNamer')
lcMySQL = loController.GetName('MySQLQuotedNamer')
messagebox('SQL Server: ' + lcMSSQL + chr(13) + 'MySQL: ' + lcMySQL)
loController.SetOriginalName('Table')
lcMSSQL = loController.GetName('MSSQLDelimitedNamer')
lcMySQL = loController.GetName('MySQLQuotedNamer')
messagebox('SQL Server: ' + lcMSSQL + chr(13) + 'MySQL: ' + lcMySQL)
```

* Attach the processor for Windows filenames and display the correct version of
* a name.

```
loController.AttachProcessor('WindowsNamer', 'windows-names.prg')
loController.SetOriginalName('SWFox: Samples')
lcName = loController.GetName('WindowsNamer')
messagebox('Windows: ' + lcName)
```

Summary

Name Syntax Checker is even more obscure than CalendarCalc but I personally can see a use for it: in Stonefield Query, we have code similar to Name Syntax Checker that deals with when to put delimiters on object names in SQL Server, MySQL, and other databases

and how to convert one of those names into a valid VFP field name. If Name Syntax Checker had been around when I wrote that code, I would've gladly used it instead.

CSVProcessor

The fourth project from António Lopes, CSVProcessor (<https://bitbucket.org/atlopes/csv>), provides CSV processing capabilities. It can do tasks such as importing from and exporting to CSV files, mapping fields in a cursor to CSV elements, handling multi-line content and date values with varying formats, handling Unicode and UTF-8 content, and auto-detection of data types.

The CSVProcessor class is defined in CSV.prg. It has numerous properties, such as ValueSeparator and DatePattern, which help you define what the CSV content looks like. It has two main methods, Import and Export, which do as their names suggest, as well as other supporting methods that may be useful. See the PEM page for documentation on this class.



CSVProcessor requires Name Syntax Checker, specifically Namer.prg and VFP-Names.prg.

Let's look at some examples of how CSVProcessor can make handling CSV files much easier.

Date handling

I've seen dates formatted many different ways in CSV files: MM/DD/YYYY, YYYY-MM-DD, YYYYMMDD, and so on. While you can custom write code to deal with that, or import the date as a string and then parse each value, that's a pain to have to do, especially if you have to import data from multiple sources, each with its own format. CSVProcessor makes it easy to handle dates: simply set the DatePattern or DateTimePattern property to a string specifying how date and datetime values are formatted. For example, "%4Y-%2M-%2D" specifies a four-digit year followed by a dash followed by a two-digit month and a dash followed by a two-digit day.

DatePattern.prg, shown in **Listing 6**, demonstrates the use of date patterns. The first time the CSV file is imported, the default datetime pattern (%4Y-%2M-%2D %2h:%2m:%2s) is used but the CSV file uses a MM/DD/YYYY HH:MM:SS format, so the datetime values are stored in a Character column. The second import uses the correct pattern so the values are placed in a DateTime column. Note also that the Import method automatically handles the header record in the CSV: it isn't added as record in the cursor as it would be with the VFP APPEND FROM command.

Listing 6. DatePattern.prg shows how specifying the date pattern makes import date values easy.

* Set the path to find all the files we need and load the libraries.

```
set path to CSVProcessor, NameSyntaxChecker  
do CSV
```

* Create a CSV file.

```
text to lcCSV noshow
ID,OccurrenceTime,Description
1,10/23/2017 11:12:40 AM,"Power off, alarm not set"
2,10/23/2017 11:13:21 AM,Restarted
3,10/23/2017 00:07:54 PM,"Signal OK, entering quiet mode"

endtext
strtofile(lcCSV, 'temp.csv')
```

* Read the CSV file using the default pattern.

```
loCSV = createobject('CSVProcessor')
loCSV.Import('temp.csv')
select (loCSV.CursorName)
browse
messagebox('Pattern: ' + loCSV.DatetimePattern + chr(13) + ;
           'Type of OccurrenceTime: ' + vartype(OccurrenceTime))
```

* Now read it again with the proper datetime format.

```
loCSV.DatetimePattern = '%2M/%2D/%4Y %2h:%2m:%2s %p'
loCSV.Import('temp.csv')
select (loCSV.CursorName)
browse
messagebox('Pattern: ' + loCSV.DatetimePattern + chr(13) + ;
           'Type of OccurrenceTime: ' + vartype(OccurrenceTime))
```

```
use
erase temp.csv
```

Appending and field mapping

The Import method can append records to an existing cursor by either passing the alias of the cursor as the second parameter to the method or by setting the Workarea property to the alias.

If the structure of the CSV file doesn't match the structure of the cursor (in my experience, it rarely does), you can specify how to map the fields in the cursor to those in the CSV file. FieldMapping is a collection, so call the Add method to specify the name of the field in the cursor. If you don't specify a second parameter, the column number in the CSV file is used, so the first call to Add is for the first column, the second call is for the second, and so on. You can also specify the name of the column as specified in the CSV header record (assuming there is one) as the second parameter, in which case the columns are matched up by name rather than position.

Listing 7, taken from FieldMapping.prg, demonstrates appending to an existing cursor and how field mapping works.

Listing 7. FieldMapping.prg shows how to append a CSV file into an existing cursor and how to map fields between the cursor and the CSV.

* Set the path to find all the files we need and load the libraries.

```
set path to CSVProcessor, NameSyntaxChecker
do CSV
```

* Create a CSV file.

```
text to lcCSV noshow
Identifier,Time,Status
2,2017-10-23 11:12:40,"Power off, alarm not set"
3,2017-10-23 11:13:21,Restarted
4,2017-10-23 00:07:54,"Signal OK, entering quiet mode"
```

```
endtext
strtofile(lcCSV, 'temp.csv')
```

* Create a cursor and import from the CSV into it. Note the order of the
* columns is different in the cursor than the CSV file so the original cursor
* is overwritten (we don't see ID 1).

```
CreateCursor()
loCSV = createobject('CSVProcessor')
loCSV.Import('temp.csv', alias())
browse
```

* Appended into an existing cursor. The columns don't match so the data is
* messed up.

```
CreateCursor()
loCSV.Workarea = 'TargetCursor'
loCSV.Import('temp.csv')
select (loCSV.CursorName)
browse
```

* This time we'll tell it how to match columns by position (the first item in
* the collection is for CVS column 1, etc.).

```
CreateCursor()
loCSV.FieldMapping.Add('ID')
loCSV.FieldMapping.Add('OccurrenceTime')
loCSV.FieldMapping.Add('Description')
loCSV.Import('temp.csv')
browse
```

* Do a mapping matching cursor and CSV column names rather than position.

```
CreateCursor()
loCSV.FieldMapping.Remove(-1)
loCSV.FieldMapping.Add('ID', 'Identifier')
loCSV.FieldMapping.Add('Description', 'Status')
loCSV.FieldMapping.Add('OccurrenceTime', 'Time')
loCSV.Import('temp.csv')
```

browse

* Unmapped fields aren't imported so that can be used to "field filter".

```
CreateCursor()  
loCSV.FieldMapping.Remove('Status')  
loCSV.Import('temp.csv')  
browse
```

```
use  
erase temp.csv
```

* Create a cursor and add a record so we can see if Import appends into the
* existing cursor.

```
function CreateCursor  
create cursor TargetCursor (ID integer, Description varchar(64), ;  
    OccurrenceTime datetime)  
insert into TargetCursor values (1, 'All systems ON', {^2017-10-23 18:00:01})
```

Encoding

CSVProcessor automatically handles encoded data. Encoding.prg, shown in **Listing 8**, demonstrates that while the VFP APPEND FROM command fails with both Unicode and UTF-8, CSVProcessor imports the data properly.

Listing 8. Encoding.prg shows that CSVProcessor automatically handles character encoding.

* Set the path to find all the files we need and load the libraries.

```
set path to CSVProcessor, NameSyntaxChecker  
do CSV
```

* Create a CSV file as Unicode with a Byte Order Mark (BOM).

```
text to lcCSV noshow  
Country, City, Population, Latitude, Longitude  
BR, Belém, 1452275, -1.455833, -48.503889  
PT, Coimbra, 143396, 40.211111, -8.429167  
ES, A Coruña, 246056, 43.365, -8.41
```

```
endtext  
strtofile(strconv(strconv(lcCSV, 1), 5), 'temp.csv', 2)
```

* First use the VFP APPEND FROM command to see what happens.

```
create cursor Temp (Country C(2), City C(10), Population N(7), ;  
    Latitude N(10, 6), Longitude N(10, 6))  
append from temp.csv type delimited  
browse  
use
```

* Import it.

```
loCSV = createobject('CSVProcessor')
loCSV.Import('temp.csv')

* Show the file to see its encoding.

modify file temp.csv

* Show the imported file.

select (loCSV.CursorName)
browse

* Now do the same but using UTF-8.

strtofile(strconv(strconv(lcCSV, 1), 9), 'temp.csv', 4)
create cursor Temp (Country C(2), City C(10), Population N(7), ;
    Latitude N(10, 6), Longitude N(10, 6))
append from temp.csv type delimited
browse
use

loCSV.Import('temp.csv')
modify file temp.csv
select (loCSV.CursorName)
browse
use

erase temp.csv
```

Exporting to CSV

Although it isn't common, sometimes you'll run across the need to create CSV files with more than one structure. I've seen this, for example, where one section is a header (such as an invoice) and another is the detail (such as invoice lines). CSVProcessor's Export function can not only export to a CSV file, it can handle appending to an existing file, so you can export from multiple cursors. You have controls over the order of the exported columns via field mapping, whether a header record is output or not, and the names of the columns in the header. Exporting.prg, shown in **Listing 9**, demonstrates this.

Listing 9. Exporting.prg shows how CSVProcessor can export multiple cursors to a single CSV file.

```
* Set the path to find all the files we need and load the libraries.

set path to CSVProcessor, NameSyntaxChecker
do CSV

* Create the first cursor.

create cursor HeaderInformation (InfoType varchar(40), InfoValue memo)
insert into HeaderInformation values ('Type', 'Music')
insert into HeaderInformation values ('Genre', 'Pop, folk, rock, fusion')
insert into HeaderInformation values ('Medium', 'Vinyl record')
```

* Create the second cursor.

```
create cursor Discography (ID_Record integer, Artist varchar(40) null, ;
    Title varchar(40) null, DateRelease date null, Duration integer)
insert into Discography values (1, 'Frank Zappa', 'Sleep Dirt', ;
    {^1979-01-19}, 2353)
insert into Discography values (2, 'The Waterboys', "Fisherman's Blues", ;
    {^1988-10-17}, 3277)
insert into Discography values (3, 'Eugenio Finardi', 'Sugo', ;
    .null., 2309)
insert into Discography values (4, 'Krafwerk', .null., ;
    {^1970-11-01}, 2379)
```

* Export the first cursor with no header row.

```
loCSV = createobject('CSVProcessor')
loCSV.Workarea = 'HeaderInformation'
loCSV.HeaderRow = .F.
loCSV.Export('temp.csv', .T.)
```

* Export the second cursor; this time, we'll specify the column names in the
* CSV file and put them in a different order than the cursor, put two blank
* lines after the records for the first cursor, and use a header row.

```
loCSV.Workarea = 'Discography'
loCSV.FieldMapping.add('Id_Record', 'ID#')
loCSV.FieldMapping.add('Title', 'Album title')
loCSV.FieldMapping.add('Artist', 'Artist(s)')
loCSV.FieldMapping.add('DateRelease', 'Release on')
loCSV.FieldMapping.add('Duration', 'Duration (in seconds)')
loCSV.HeaderRow = .T.
loCSV.SkipRows = 2
loCSV.Export('temp.csv', .T., .T.)
```

* Show the result.

```
modify file temp.csv
erase temp.csv
```

Summary

CSVProcessor is a very useful class, as importing CSV data into VFP has numerous complications, such as date format, column order, and character encoding, that make a seemingly easy process using APPEND FROM far more complicated than you think.

XML Library Set

The last project from António Lopes is XML Library Set, available at <https://bitbucket.org/atlopes/xml>. It provides classes for processing XML. Its three main classes are:

- XMLSerializer (xml-serializer.prg).
- XMLCanonicalizer (xml-canonicalizer.prg).

- XMLSampler (xml-sampler.prg).

Documentation for all three is available in appropriately named Markdown files.



XML Library Set requires Name Syntax Checker, specifically Namer.prg, XML-Names.prg, and VFP-Names.prg.

XMLSerializer

Sometimes it's handy to serialize a VFP object; that is, save the values of its properties so they can be later restored to the same values. One example is a configuration object that stores settings such as the size and position of the main form and what font to use for controls. XMLSerializer has VFPToXML and XMLtoVFP methods that do just that.

VFPToXML is straightforward: pass it an object to serialize and the name of the root element in the XML. It returns an XML DOM object containing the XML for the object.

Listing 10, taken from TestXMLSerializer.prg, shows how to use this method.

Listing 10. VFPToXML serializes a VFP object.

* Create an XMLSerializer object and serialize the configuration object.

```
loConfiguration = createobject('Configuration')
loSerializer     = createobject('XMLSerializer')
loXMLDOM         = loSerializer.VFPToXML(loConfiguration, 'configuration')
strtofile(loXMLDOM.xml, 'config.xml')
```

```
define class Configuration as Custom
    BackColor = rgb(255, 255, 255)
    ForeColor = rgb( 0, 0, 0)
    FontName  = 'Segoe UI'
    FontSize  = 10
enddefine
```

Here's what the XML looks like (formatted here with line breaks and tabs to make it easier to read):

```
<?xml version="1.0"?>
<configuration>
  <BACKCOLOR><![CDATA[16777215]]></BACKCOLOR>
  <FONTNAME><![CDATA[Segoe UI]]></FONTNAME>
  <FONTSIZE><![CDATA[10]]></FONTSIZE>
  <FORECOLOR><![CDATA[0]]></FORECOLOR>
</configuration>
```

To deserialize the object, pass the XML to the XMLtoVFP method. Note that the object returned is based on Empty rather than the class the original object was based on. Also, it has a different hierarchy than the original object: it has a member with the same name as

the root name specified when you serialized the object, and that object has the properties of the original object. For example, deserializing the XML generated by the code in Listing 10 creates an object with a Configuration property, which contains an object with BackColor, ForeColor, FontName, and FontSize properties. There are also numerous properties in both objects that start with "XML;" these contain serialization settings. If you want a simpler object without those properties, pass the GetSimpleCopy method the deserialized object. **Listing 11** shows code also taken from TestXMLSerializer.prg that demonstrates this.

Listing 11. XMLtoVFP deserializes an object.

```
lcXML          = filetostr('config.xml')
loXMLObject    = loSerializer.XMLtoVFP(lcXML)
loConfig       = loSerializer.GetSimpleCopy(loXMLObject)
loConfiguration = loConfig.Configuration
messagebox(loConfiguration.FontName)
```

XMLSerializer can handle more complex objects. The code in **Listing 12** creates a Customer object which has an array of Contact objects (I first tried using a collection rather than an array but XMLSerializer doesn't currently support a collection). Again, be aware that the deserialized object isn't an instance of Customer nor are the objects in the Contacts array instances of Contact; they're Empty objects with the appropriate properties added.

Listing 12. XMLSerializer can support complex objects as well.

```
loCustomer = createobject('Customer')
loCustomer.CompanyName = 'Geek Gatherings'
loCustomer.AddContact('Rick Schummer')
loCustomer.AddContact('Tamar Granor')
loCustomer.AddContact('Doug Hennig')
loXMLDOM = loSerializer.VFPToXML(loCustomer, 'customer')
lcXML     = loXMLDOM.xml
```

```
loXMLObject = loSerializer.XMLtoVFP(lcXML)
loCust      = loSerializer.GetSimpleCopy(loXMLObject)
loCustomer = loCust.Customer
messagebox(loCustomer.Contacts[1].ContactName)
```

```
define class Configuration as Custom
    BackColor = rgb(255, 255, 255)
    ForeColor = rgb( 0, 0, 0)
    FontName = 'Segoe UI'
    FontSize = 10
enddefine
```

```
define class Customer as Custom
    CompanyName = ''
    dimension Contacts[1]

    function Init
        This.Contacts[1] = null
```



```
endfunc

function AddContact(tcName)
    loContact = createobject('Contact')
    loContact.ContactName = tcName
    lnContacts = alen(This.Contacts)
    if not isnull(This.Contacts[1])
        lnContacts = lnContacts + 1
    endif not isnull(This.Contacts[1])
    dimension This.Contacts[lnContacts]
    This.Contacts[lnContacts] = loContact
endfunc
enddefine

define class Contact as Custom
    ContactName = ''
enddefine
```

XMLSerializer has other methods; see the XML-Serializer page in the repository for details.

XMLCanonicalizer and XMLSampler

XMLCanonicalizer converts XML into W3C canonical form. XMLSampler creates sample XML that matches the specified schema. I don't see a lot of use for these classes so I didn't look at them in any detail except reviewing the documentation in the appropriate pages in the repository.

Summary

XMLSerializer may be useful if you need to serialize an object and aren't concerned that when you deserialize it, you end up with something based on Empty (that is, you aren't going to call any methods of the class, just access its properties). The other classes in this library are very specialized so check out the documentation to see if they may of any use to you.

Cursor and Connector Properties Viewer

Cursor and Connector Properties Viewer (<https://github.com/VFPX/CCPropsViewer>) is a library by Martina Jindrová that allows you to display and set cursor and connection properties. It works with any cursor, whether opened from a table, local view, remote view, or CursorAdapter. It can be used either at runtime or in the IDE as a debugging aid.

As you can see in **Figure 13**, it displays the various properties of a cursor. In this case, the cursor comes from a remote view, so it shows that as well as the name of the database the view is defined in, the current buffer mode, the SQL statement, the update settings, the connection settings, and other view settings such as FetchSize and MaxRecords. You can change the values of some of these settings.

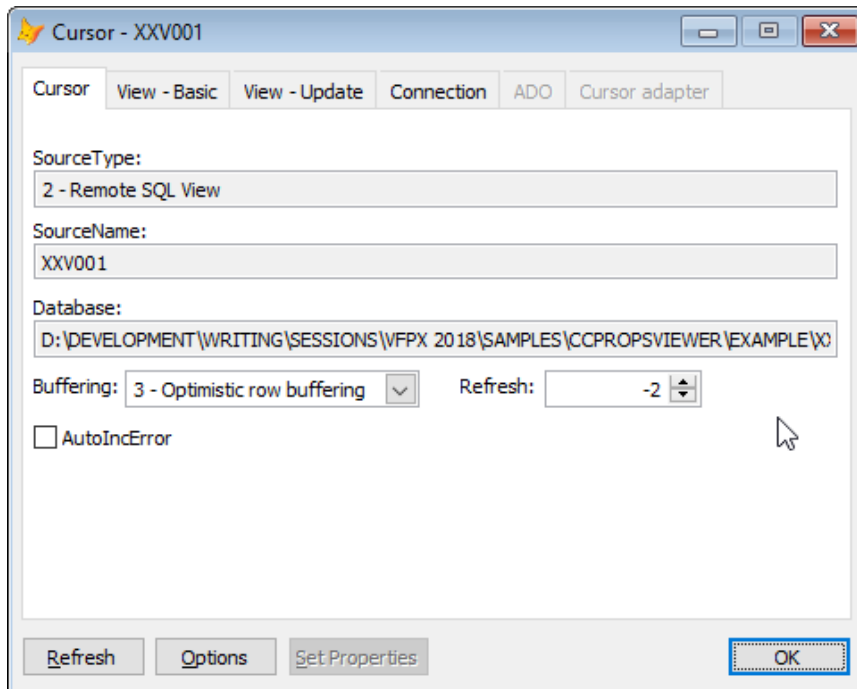


Figure 13. Cursor and Connector Properties Viewer displays the settings of cursors.

Summary

Cursor and Connector Properties Viewer is a useful tool for debugging cursors, especially those created from views or CursorAdapters. Rather than issuing a series of CURSORGETPROP statements in DEBUG OUT or MESSAGEBOX, you can easily see the settings, and even change them, visually.

DBC Low Level Functions

DBC Low Level Functions (<https://github.com/VFPX/DBCLowLevelFunctions>) is another library by Martina Jindrová. As its name suggests, this one allows you to read from and write to values of all properties and objects without opening a database with OPEN DATABASE. This could serve as the starting point for your own DBC-related utilities.

The library consists of a single program, DBC.prg, along with an accompanying include file, DBC.h. Documentation is provided in the Doc folder

Start by instantiating `_DBC_Check` in DBC.prg, then call `OpenTable` to open the specified database as a table using the specified alias:

```
set procedure to dbc.prg
lcDBC = 'MyDatabase.dbc'
lcAlias = sys(2015)
loDBC = createobject('_DBC_Check')
if loDBC.OpenTable(lcDBC, lcAlias, '')
    wait window 'Cannot open DBC.'
    return
endif
```

You can then call various methods:

- EnumObjects fills an array with the names of all objects or just certain types of objects, such as tables or fields in a specific view.
- EnumProperty fills an array with the property values for the specified object.
- GetProperty reads the value of the specified property for the specified object.
- SetProperty saves the value of the specified property for the specified object.
- ClearProperty removes the specified property for the specified object.
- GetSP reads the stored procedures.
- SetSP saves the specified code as the stored procedures.
- GetIDObject gets the DBC's OBJECTID value for the specified object name.
- RIInfo gets the referential integrity information for a specific relation.

See the sample DBCC.prg and DBCT.prg programs that come with the project for examples of its use.

One of the interesting things about this project is that it allows you to change the values of properties that VFP considers read-only, such as the Path and PrimaryKey properties of a table. This is useful if, for example, you need to manually move things around and can't set these properties the usual way.

Summary

DBC Low Level Functions is a very specialized project. It's one of those things that you normally won't need but when you need it, you really need it.

FoxTypes

FoxTypes (<https://github.com/eselje/FoxTypes>) is a new project by Eric Selje that makes VFP data types act like .NET data types. That is, it treats them as objects with properties and methods rather than functions that act on data types. For example, to get the current date and time in UTC (Coordinated Universal Time), you can use:

```
loValue = createobject('DateTime')
loValue.Date = date()
ltUTC = loValue.UTCNow
```

The DateTime class has properties and methods that closely match those in the .NET DateTime class, making it easy to convert .NET code working with DateTime values into VFP code. For example, the AddDays function adds the specified number of days to the DateTime value.

The most useful method is the Format method of the String class, which mimics the static String.Format method in .NET. It takes a format string followed by up to 10 parameters of

any data types and returns a formatted string with the values of the parameters inserted. The format string uses parameter numbers (zero-based) surrounded by curly braces as place holders. The place holder can also contain a format specifier. For example:

```
String = createobject('String')
lcMessage = String.Format('Today is {0:D}. There are {1:c2} invoices to process', ;
    datetime(), lnInvoices)
```

The format string specifies that the two value parameters are inserted into the indicated places in the string with the first (remember, it's zero-based so {0} means the first value parameter and {1} means the second) formatted as a long date (the "d") and the second as a currency value (the "c") with two decimal places (the "2").

Another useful method is `String.NumberToWords`, which converts a numeric value to spelled-out English words. For example:

```
messagebox(String.NumberToWords(1234.56))
    && displays One Thousand Two Hundred Thirty Four and 56/100
```

See the `FoxTypes` documentation for the complete list of properties and methods in the classes.



As of this writing, a lot of the properties and methods in the `FoxTypes` classes haven't been implemented yet.

Summary

`FoxTypes` has some useful methods now but will be a lot more useful once the classes are finished. In addition to providing the `Format` and `NumberToWords` methods, you may find `FoxType` useful if you'd rather work with properties and methods rather than functions or need to translate .NET code into its VFP equivalent.

FastXTab

`FastXTab` (<https://github.com/VFPX/FastXTab>) is a replacement for `VFPXTab` which comes with VFP. Its purpose is to create a "cross-tabulation" cursor (similar to a `PivotTable` in Microsoft Excel) that can be used for reporting. `FastXTab` is a lot faster than `VFPXTab` and has a lot more functionality.

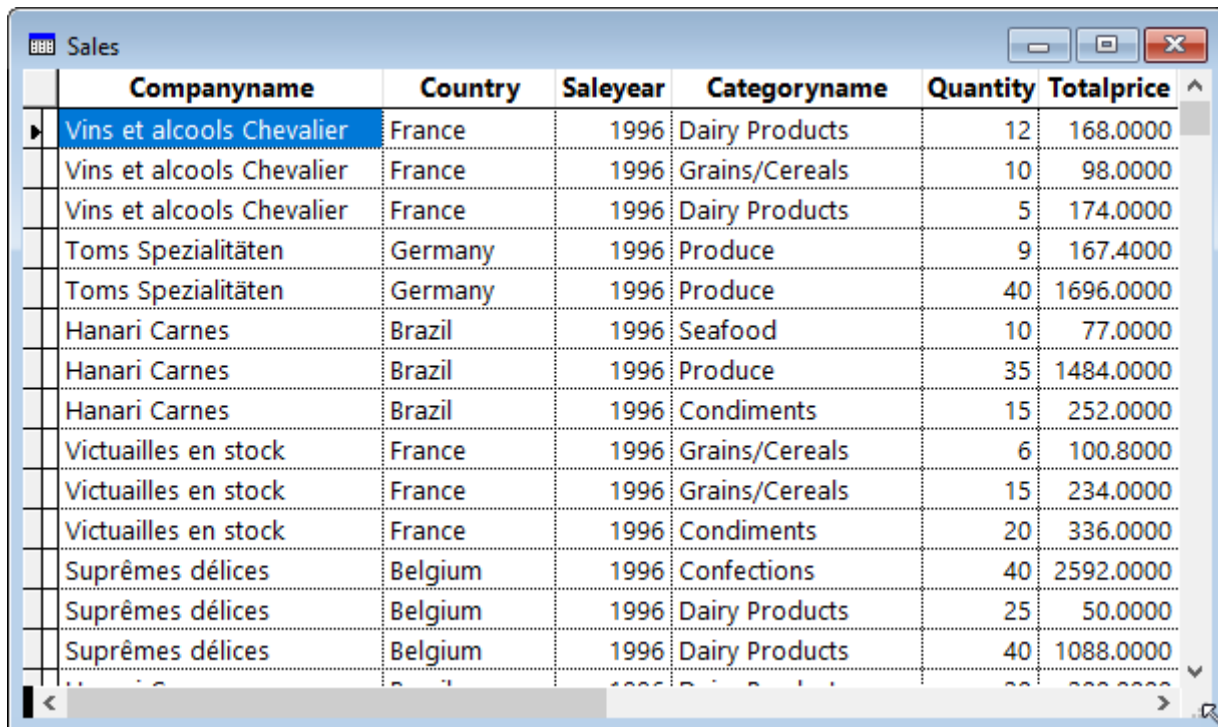
`FastXTab` was originally created by Alexander Golovlev and improved by Vilhelm-Ion Praisach. There are versions for VFP 9 and VFP 6. If you attended Southwest Fox 2017, Tamar Granor's *Turning Data Sideways: Crosstabs and Pivot Tables* session discussed `FastXTab` at length.

The `FastXTab` class, defined in `FastXTab.prg`, processes a cursor with at least three columns. By default, the data in the first column becomes the rows in the result, the data in the second column becomes the columns, and the data in the third column is aggregated

(summed by default) to form the data. However, the source cursor can be more complex: FastXTab only processes the fields you tell it to by setting various properties such as cRowField, cColField, and cDataField.

Let's start looking at FastXTab by creating a cursor of sales from the VFP sample Northwind database (the code shown here was taken from TestFastXTab.prg which accompanies this document). **Figure 14** shows what the result looks like.

```
open database (addbs(_samples) + 'Northwind\Northwind')
select Customers.CompanyName, ;
       Customers.Country, ;
       year(Orders.OrderDate) as SaleYear, ;
       Categories.CategoryName, ;
       OrderDetails.Quantity, ;
       OrderDetails.UnitPrice * OrderDetails.Quantity as TotalPrice ;
from Customers ;
inner join Orders on Customers.CustomerID = Orders.CustomerID ;
inner join OrderDetails on Orders.OrderID = OrderDetails.OrderID ;
inner join Products on OrderDetails.ProductID = Products.ProductID ;
inner join Categories on Products.CategoryID = Categories.CategoryID ;
into cursor Sales
```



Companyname	Country	Saleyear	Categoryname	Quantity	Totalprice
Vins et alcools Chevalier	France	1996	Dairy Products	12	168.0000
Vins et alcools Chevalier	France	1996	Grains/Cereals	10	98.0000
Vins et alcools Chevalier	France	1996	Dairy Products	5	174.0000
Toms Spezialitäten	Germany	1996	Produce	9	167.4000
Toms Spezialitäten	Germany	1996	Produce	40	1696.0000
Hanari Carnes	Brazil	1996	Seafood	10	77.0000
Hanari Carnes	Brazil	1996	Produce	35	1484.0000
Hanari Carnes	Brazil	1996	Condiments	15	252.0000
Victuailles en stock	France	1996	Grains/Cereals	6	100.8000
Victuailles en stock	France	1996	Grains/Cereals	15	234.0000
Victuailles en stock	France	1996	Condiments	20	336.0000
Suprêmes délices	Belgium	1996	Confections	40	2592.0000
Suprêmes délices	Belgium	1996	Dairy Products	25	50.0000
Suprêmes délices	Belgium	1996	Dairy Products	40	1088.0000

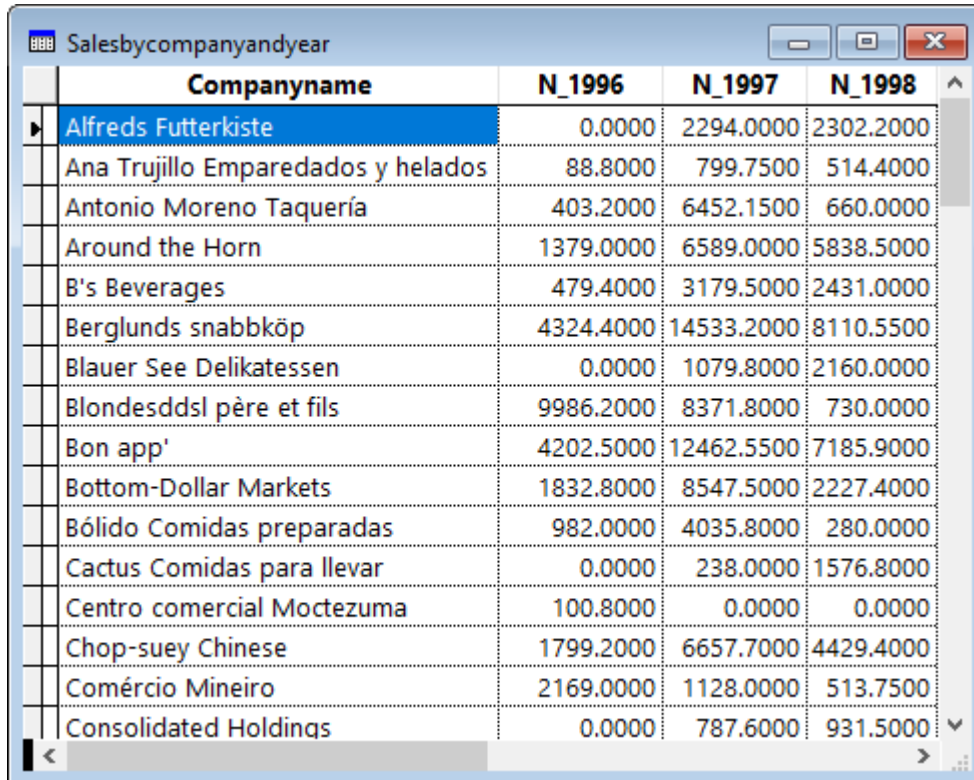
Figure 14. The cursor used as the source for creating cross-tab cursors.

The first example creates a cross-tab cursor showing sales by company and year, displaying zero rather than null. The result is shown in **Figure 15**. Notice there's only one row per company (which the cursor is sorted on) and separate columns for each year, named "N_" followed by the year since a value like "1996" isn't a valid field name.

```

loXTab = newobject('FastXTab', 'FastXTab.prg')
loXTab.cRowField = 'CompanyName'
loXTab.cColField = 'SaleYear'
loXTab.cDataField = 'TotalPrice'
loXTab.lCloseTable = .F.
loXTab.lCursorOnly = .T.
loXTab.lBrowseAfter = .T.
loXTab.lDisplayNulls = .F.
loXTab.cOutFile = 'SalesByCompanyAndYear'
loXTab.RunXTab()

```



Companyname	N_1996	N_1997	N_1998
Alfreds Futterkiste	0.0000	2294.0000	2302.2000
Ana Trujillo Emparedados y helados	88.8000	799.7500	514.4000
Antonio Moreno Taquería	403.2000	6452.1500	660.0000
Around the Horn	1379.0000	6589.0000	5838.5000
B's Beverages	479.4000	3179.5000	2431.0000
Berglunds snabbköp	4324.4000	14533.2000	8110.5500
Blauer See Delikatessen	0.0000	1079.8000	2160.0000
Blondesddsl père et fils	9986.2000	8371.8000	730.0000
Bon app'	4202.5000	12462.5500	7185.9000
Bottom-Dollar Markets	1832.8000	8547.5000	2227.4000
Bólido Comidas preparadas	982.0000	4035.8000	280.0000
Cactus Comidas para llevar	0.0000	238.0000	1576.8000
Centro comercial Moctezuma	100.8000	0.0000	0.0000
Chop-suey Chinese	1799.2000	6657.7000	4429.4000
Comércio Mineiro	2169.0000	1128.0000	513.7500
Consolidated Holdings	0.0000	787.6000	931.5000

Figure 15. A cross-tab cursor showing sales by company and year.

The next one does the same thing but displays nulls. Note that we select the Sales cursor first since the selected workarea has to contain the source data. Also note that since we're reusing the same FastXTab object, we only have to change the things that are different from the previous run. The result is the same as Figure 15 except ".NULL." displays in place of "0.0000."

```

select Sales
loXTab.lDisplayNulls = .T.
loXTab.RunXTab()

```

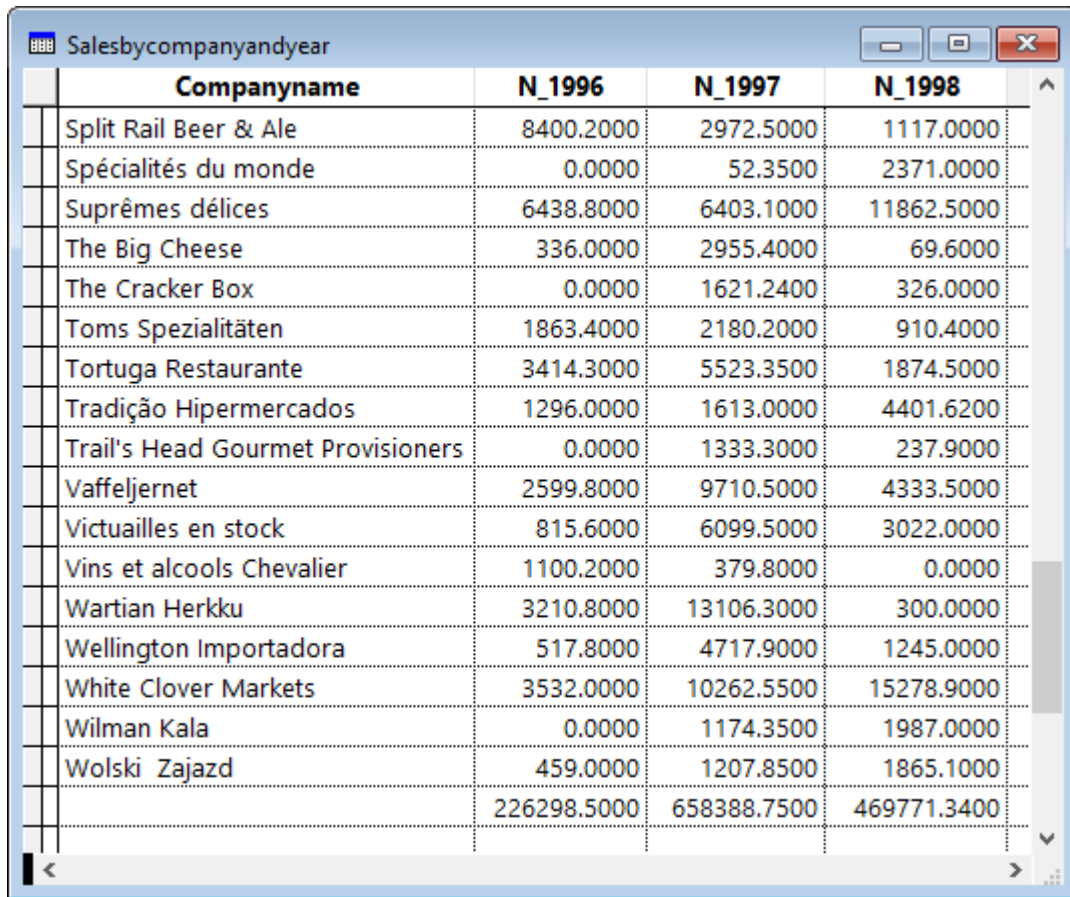
Let's add a totals row to the results. Notice the totals in the last row in **Figure 16**.

```

select Sales
loXTab.lDisplayNulls = .F.
loXTab.lTotalRows = .T.

```

```
loXTab.RunXTab()
```



	Companyname	N_1996	N_1997	N_1998
	Split Rail Beer & Ale	8400.2000	2972.5000	1117.0000
	Spécialités du monde	0.0000	52.3500	2371.0000
	Suprêmes délices	6438.8000	6403.1000	11862.5000
	The Big Cheese	336.0000	2955.4000	69.6000
	The Cracker Box	0.0000	1621.2400	326.0000
	Toms Spezialitäten	1863.4000	2180.2000	910.4000
	Tortuga Restaurante	3414.3000	5523.3500	1874.5000
	Tradição Hipermercados	1296.0000	1613.0000	4401.6200
	Trail's Head Gourmet Provisioners	0.0000	1333.3000	237.9000
	Vaffeljernet	2599.8000	9710.5000	4333.5000
	Victuailles en stock	815.6000	6099.5000	3022.0000
	Vins et alcools Chevalier	1100.2000	379.8000	0.0000
	Wartian Herkku	3210.8000	13106.3000	300.0000
	Wellington Importadora	517.8000	4717.9000	1245.0000
	White Clover Markets	3532.0000	10262.5500	15278.9000
	Wilman Kala	0.0000	1174.3500	1987.0000
	Wolski Zajazd	459.0000	1207.8500	1865.1000
		226298.5000	658388.7500	469771.3400

Figure 16. A cross-tab cursor with a totals row.

Let's show multiple data fields: both the quantity sold and total price. `nMultiDataField` specifies the number of data fields and the `acDataField` array specifies the source fields for the data fields. In **Figure 17**, the first column for each year (for example, "N_1997") is the sum of the quantity and the second column (for example, "N_1997_2") is the sum of the total price.

```
select Sales
loXTab.nMultiDataField = 2
loXTab.acDataField[1] = 'Quantity'
loXTab.acDataField[2] = 'TotalPrice'
loXTab.RunXTab()
```

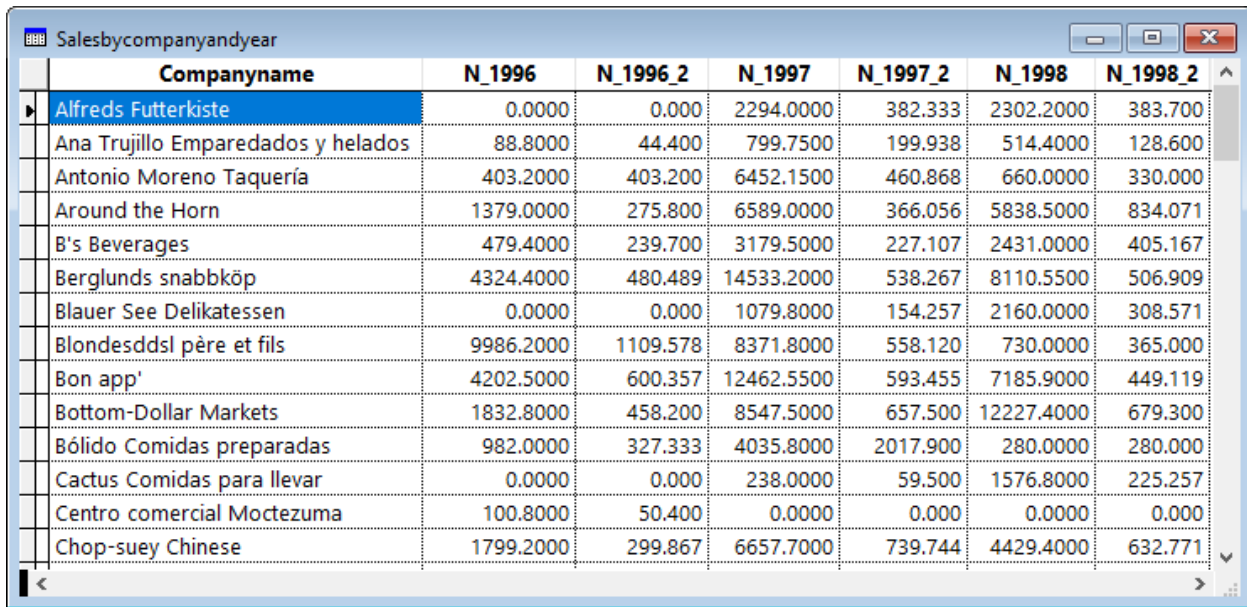
Companyname	N_1996	N_1996_2	N_1997	N_1997_2	N_1998	N_1998_2
Alfreds Futterkiste	0	0.0000	79	2294.0000	95	2302.2000
Ana Trujillo Emparedados y helados	6	88.8000	28	799.7500	29	514.4000
Antonio Moreno Taquería	24	403.2000	295	6452.1500	40	660.0000
Around the Horn	105	1379.0000	371	6589.0000	174	5838.5000
B's Beverages	39	479.4000	165	3179.5000	89	2431.0000
Berglunds snabbköp	169	4324.4000	481	14533.2000	351	8110.5500
Blauer See Delikatessen	0	0.0000	68	1079.8000	72	2160.0000
Blondesddsl père et fils	268	9986.2000	348	8371.8000	50	730.0000
Bon app'	181	4202.5000	486	12462.5500	313	7185.9000
Bottom-Dollar Markets	81	1832.8000	454	8547.5000	421	12227.4000
Bólido Comidas preparadas	90	982.0000	60	4035.8000	40	280.0000
Cactus Comidas para llevar	0	0.0000	20	238.0000	95	1576.8000
Centro comercial Moctezuma	11	100.8000	0	0.0000	0	0.0000
Chop-suey Chinese	122	1799.2000	219	6657.7000	124	4429.4000
Comércio Mineiro	60	2169.0000	45	1128.0000	28	513.7500
Consolidated Holdings	0	0.0000	54	787.6000	33	931.5000

Figure 17. A cross-tab cursor with multiple data fields.

Let's show both the sum and average of the total price. The `anFunctionType` array has the same number of columns as `acDataField`, so the first element is for the first data field, the second one for the second, and so on. A value of 1 means sum, 2 means Count, 3 means Average, 4 means min, 5 means max, and 6 means use a custom function (see the `FastXTab` documentation for details). In **Figure 18**, the first column for each year (for example, "N_1997") is the sum of the total price and the second column (for example, "N_1997_2") is the average.

```
#define cnSUM      1
#define cnAVERAGE 3

select Sales
loXTab.nMultiDataField = 2
loXTab.acDataField[1] = 'TotalPrice'
loXTab.anFunctionType[1] = cnSUM
loXTab.acDataField[2] = 'TotalPrice'
loXTab.anFunctionType[2] = cnAVERAGE
loXTab.RunXTab()
```

Companyname	N_1996	N_1996_2	N_1997	N_1997_2	N_1998	N_1998_2
Alfreds Futterkiste	0.0000	0.000	2294.0000	382.333	2302.2000	383.700
Ana Trujillo Emparedados y helados	88.8000	44.400	799.7500	199.938	514.4000	128.600
Antonio Moreno Taqueria	403.2000	403.200	6452.1500	460.868	660.0000	330.000
Around the Horn	1379.0000	275.800	6589.0000	366.056	5838.5000	834.071
B's Beverages	479.4000	239.700	3179.5000	227.107	2431.0000	405.167
Berglunds snabbköp	4324.4000	480.489	14533.2000	538.267	8110.5500	506.909
Blauer See Delikatessen	0.0000	0.000	1079.8000	154.257	2160.0000	308.571
Blondesddsl père et fils	9986.2000	1109.578	8371.8000	558.120	730.0000	365.000
Bon app'	4202.5000	600.357	12462.5500	593.455	7185.9000	449.119
Bottom-Dollar Markets	1832.8000	458.200	8547.5000	657.500	12227.4000	679.300
Bólido Comidas preparadas	982.0000	327.333	4035.8000	2017.900	280.0000	280.000
Cactus Comidas para llevar	0.0000	0.000	238.0000	59.500	1576.8000	225.257
Centro comercial Moctezuma	100.8000	50.400	0.0000	0.000	0.0000	0.000
Chop-suey Chinese	1799.2000	299.867	6657.7000	739.744	4429.4000	632.771

Figure 18. This cross-tab cursor shows the sum and the average of the total price in the data fields.

The next example creates a cross-tab cursor by category and year with country as the page field; see **Figure 19**.

```
select Sales
loXTab = newobject('FastXTab', 'FastXTab.prg')
loXTab.cPageField = 'Country'
loXTab.cRowField = 'CategoryName'
loXTab.cColField = 'SaleYear'
loXTab.cDataField = 'TotalPrice'
loXTab.lCloseTable = .F.
loXTab.lCursorOnly = .T.
loXTab.lBrowseAfter = .T.
loXTab.cOutFile = 'SalesByCountryCategoryYear'
loXTab.RunXTab()
```

	Country	Categoryname	N_1996	N_1997	N_1998
▶	Argentina	Beverages	0.0000	54.0000	1744.0000
	Argentina	Condiments	0.0000	285.0000	622.0000
	Argentina	Confections	0.0000	370.4000	1764.7000
	Argentina	Dairy Products	0.0000	122.5000	1021.0000
	Argentina	Grains/Cereals	0.0000	0.0000	390.0000
	Argentina	Produce	0.0000	594.2000	544.8000
	Argentina	Seafood	0.0000	390.5000	216.0000
	Austria	Beverages	16192.8000	5649.2500	4610.0000
	Austria	Condiments	3010.5000	10124.4000	3667.5000
	Austria	Confections	874.0000	11454.8500	2324.5000
	Austria	Dairy Products	3182.4000	12379.0000	14781.5000
	Austria	Grains/Cereals	1321.0000	4954.0000	8579.2500
	Austria	Meat/Poultry	1386.0000	9055.4800	1560.0000
	Austria	Produce	1161.2000	8862.7500	3732.0000
	Austria	Seafood	2224.1000	672.2500	7737.9000

Figure 19. A cross-tab cursor with a page field.

Rather than creating a source cursor with the desired calculations, we can make FastXTab do some of the work. In this case, we'll just grab the order date rather than using the YEAR() function on it; instead, we'll tell FastXTab to use YEAR(OrderDate) for the columns. The result looks the same as Figure 15.

```
select Customers.CompanyName, ;
       Customers.Country, ;
       Orders.OrderDate, ;
       Categories.CategoryName, ;
       OrderDetails.Quantity, ;
       OrderDetails.UnitPrice * OrderDetails.Quantity as TotalPrice ;
from Customers ;
inner join Orders on Customers.CustomerID = Orders.CustomerID ;
inner join OrderDetails on Orders.OrderID = OrderDetails.OrderID ;
inner join Products on OrderDetails.ProductID = Products.ProductID ;
inner join Categories on Products.CategoryID = Categories.CategoryID ;
into cursor Sales
```

```
loXTab = newobject('FastXTab', 'FastXTab.prg')
loXTab.cRowField   = 'CompanyName'
loXTab.cColField   = 'year(OrderDate)'
loXTab.cDataField  = 'TotalPrice'
loXTab.lCloseTable = .F.
loXTab.lCursorOnly = .T.
loXTab.lBrowseAfter = .T.
loXTab.cOutFile    = 'SalesByCompanyAndYear'
loXTab.RunXTab()
```

FastXTab has a lot more capabilities: see Tamar's white paper from her 2017 session, the sample programs that come with FastXTab, and the documentation in the repository.

Summary

FastXTab is a very useful tool for creating cursors suitable for reporting, such as those you might use with FoxCharts (discussed next). We use it in Stonefield Query when outputting cross-tab reports to Microsoft Excel PivotTables.

FoxCharts

FoxCharts is a very cool library: it provides beautiful charts and graphs to your VFP applications using 100% VFP code. Creating even complex charts is relatively easy and you have a great deal of control over the appearance and behavior of the charts. Although there are a lot of properties to learn, the documentation is excellent.

However, the FoxCharts project (<https://github.com/VFPX/FoxCharts>) is frankly a bit of a mess. There are three separate versions of FoxCharts in that repository:

- FoxCharts_1.37b: the latest version created by Cesar Chalom.
- FoxCharts1.46_Beta: a newer version updated by Alex Yudin.
- FoxCharts1.47b: an even newer version from Alex.

I'm not sure why Alex split 1.47 off from 1.46 rather than just updating 1.46 to 1.47. However, since 1.47 is newer, that's likely the one you'll want to use if you want the new features Alex added. The list of new features up to 1.46.1 is included in the README.md for the project, while the README.md in the FoxCharts1.47b folder has the 1.47 changes (although as of this writing there isn't much there).

Let's look at some of the new features.

- There are several new bar types (the BarType property): cone (3), pyramid (4), conoid (a cone with the top cut off, 5), and frustrum (a pyramid with the top cut off, 6).
- There are a couple of new brush types (the BrushType property): texture (4) and colored hatch (5). A texture brush uses images to paint with; set the new TextureTheme property to the name of a subdirectory of the Textures folder containing the images to use. The existing hatch brush (3) is monochrome; if you set BrushType to 5, FoxCharts uses the color for the data point as the color of the brush.
- The new GradientShapeDirection property determines the gradient direction for gradient brush charts: 0 = Horizontal, 1 = Vertical, 2 = Diagonal1, and 3 = Diagonal 2.
- The new GradientVertCount and GradientHorCount properties determine the Shape Gradient Brush stripes count.
- Clicking an item in the legend draws lines from the legend to each data point that the clicked item represents; see **Figure 20**.

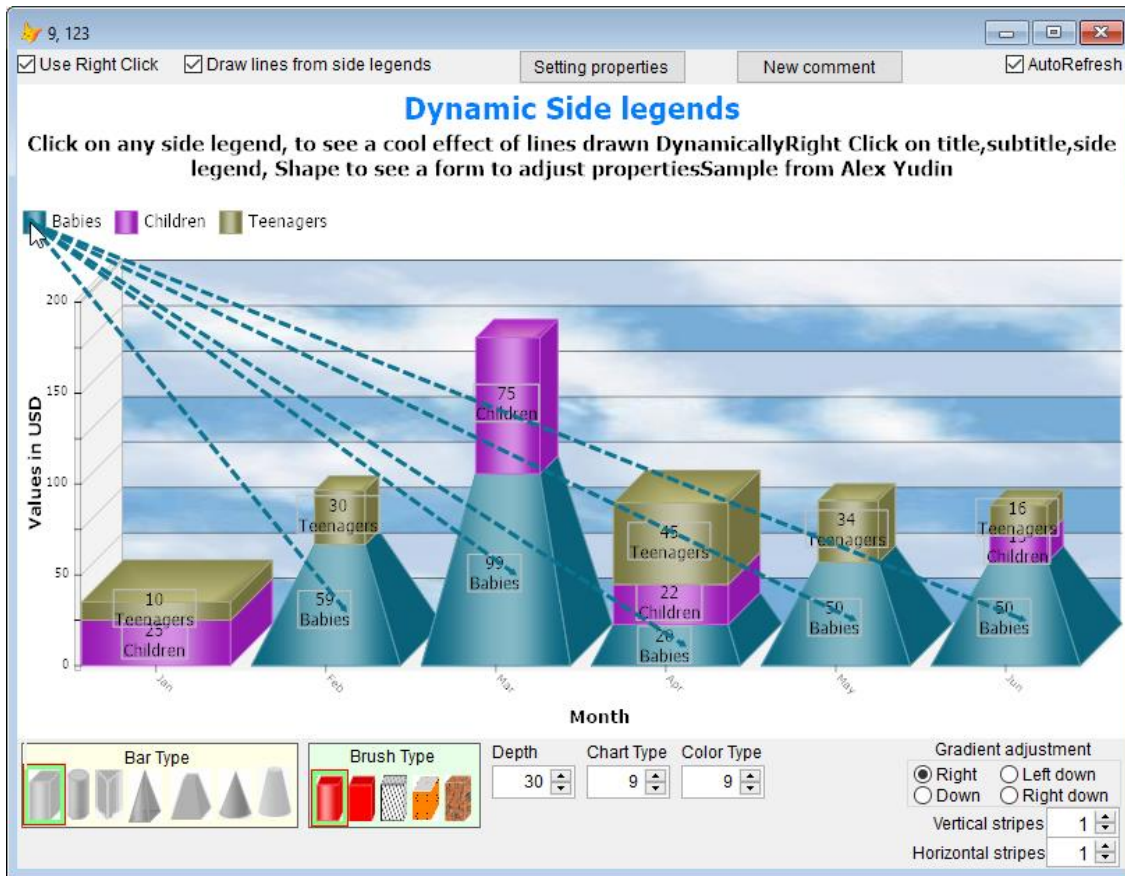


Figure 20. Clicking an item in the legend draws lines to data points for that item.

- A new editor form, shown in **Figure 21**, allows you to change more properties of the chart visually. Right-click the chart to display the form.

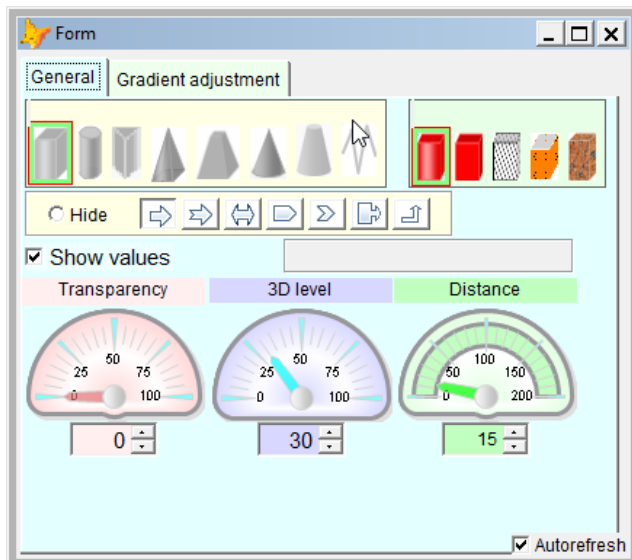


Figure 21. This chart editor form is accessed by right-clicking the chart.

To check out some of these new features, run ChartsSamples_v1_4x.scx in the SamplesBeta folder.

Summary

FoxCharts is one of the most useful VFPX project available and the latest update from Alex Yudin have made it even better. There are a few bugs in Alex's code but they should be easy to fix.

Dynamic Forms

Matt Slay's Dynamic Forms project (<https://github.com/mattslay/DynamicForms>) creates forms on the fly using markup code similar in purpose, although not in syntax, to XAML or HTML. I was initially uncertain whether I would make use of Dynamic Forms myself; after all, it's not that hard to create a form for a specific task. However, I came across the need to ask the user for input about things that can change from user to user or application to application, and Dynamic Forms soon became a frequently used tool in my toolbox.

Let's look at some advanced uses of Dynamic Form, especially data driving the creating of the markup code for the forms. Although you may certainly use the code that accompanies this document as is, I expect it will more likely serve as a source of inspiration for you to create your own dynamic forms.



The code for this document uses a version of DynamicForm.prg that I've customized a bit. I've passed on the changes I made to Matt but also included my copy of DynamicForm.prg with the sample file accompanying this document.

Custom fields

Some applications were built to be customized. For example, GoldMine and Act!, two popular customer relationship management (CRM) systems, both allow users to add custom fields to their tables. The reason is that often organizations want to record additional details about customers beyond the standard things like company name and address, such as type of customer and what month their yearend is. Both programs allow the user to specify the field name, data type, size, caption, and other meta data for the custom fields, and provide a data entry form in which the user can enter the values of those custom fields for a customer.

Let's look at doing something similar in a VFP application. We'll start with FieldDef.dbf, a table which contains the definitions of the custom fields. Its structure is shown in **Table 2**. As you can see, FieldDef contains both the definition of the custom fields (the actual custom field values are stored in CustomFields.dbf) and the specifications for the controls used to edit the values of the fields. **Figure 22** shows the records in the copy of FieldDef.dbf included in the code for this document.

Presumably, the application includes a data entry form that allows the user to define new custom fields, which adds records to FieldDef and new fields to CustomFields. We won't look at that part, just how they can edit the values of the custom fields, since this document is about creating dynamic forms.

Table 2. The structure of FieldDef.dbf.

Name	Type	Purpose
Order	I	The order of the field in the data entry form
Name	C(30)	The field name
DataType	C(1)	The field data type
Length	I	The field length
Decimals	I	The number of decimals
Caption	C(60)	The caption
Class	C(30)	The class for the control used to edit the field; empty to use a class based on DataType, such as a textbox for character fields
Library	C(30)	The VCX containing the class
Top	I	The top position of the control; 0 to auto-position
Left	I	The left position of the control; 0 to auto-position
Height	I	The height of the control; 0 for auto
Width	I	The width of the control; 0 for auto
Anchor	I	The Anchor setting for the control
Properties	M	The values of properties for the control using Dynamic Form syntax
Events	M	A comma-delimited list of events to handle for the control
Code	M	The event code to execute

Order	Name	Datatype	Length	Decimals	Caption
10	SendEmail	L	1	0	Can email
20	MaintRem	L	1	0	Send software maintenance reminder
30	Type	N	1	0	Type
40	SalesRep	C	10	0	Sales rep
50	YearEnd	N	2	0	Year end

Figure 22. The FieldDef table contains the definitions of the custom fields and the controls used to edit them.

The CustomFields class in FieldControls.vcx is a simple form class used to edit the values of the custom fields for the specified customer. It has OK and Cancel buttons and an instance of the SFDynamicRender class, but no other controls. The Init method (**Listing 13**) accepts as parameters the name of the current customer and the customer's ID. It finds the customer's record in CustomFields.dbf, creating one if necessary. It then uses SCATTER to create a data object for the record (the properties of this object are the ControlSource for the controls that Dynamic Form adds to the form), opens the FieldDef table, and asks the SFDynamicRender object on the form to render the controls for the custom fields. We'll look at SFDynamicRender later.

Listing 13. The Init method of the CustomFields class.

```
lparameters tcCustomerName, ;
    tiCustNo
This.Caption = 'Custom Fields for ' + tcCustomerName

* Open the CUSTOMFIELDS table and find the record for the specified customer,
* creating it if necessary.

if used('CUSTOMFIELDS')
    select CUSTOMFIELDS
    set order to CUSTNO
else
    select 0
    use CUSTOMFIELDS order CUSTNO again shared
endif used('CUSTOMFIELDS')
seek tiCustNo
if not found()
    insert into CUSTOMFIELDS (CUSTNO) ;
        values (tiCustNo)
endif not found()

* Create a data object from the record for the rendering engine.

scatter memo name This.oDataObject
This.oRender.oDataObject = This.oDataObject

* Render controls for the custom fields.

select 0
use FIELDDEF order ORDER again shared
This.oRender.Render(This)
use
```

The Click methods of the OK and Cancel buttons close the form, but OK first uses GATHER to save the properties of the data object to the record in CustomFields.

SFDynamicRender, contained in SFDynamicForm.vcx, is a Custom subclass that uses Dynamic Form to render the controls defined in a table or cursor in the current workarea. It isn't specific for working with custom fields; we'll use this same class later for a different purpose. It can be used for any data-driven form as long as the structure of the table or cursor is similar to FieldDef.dbf (DataType, Length, and Decimals aren't used but the other fields are) and as long as the form it's used with has an oDataObject property containing an object with properties matching the contents of the Name field in the table or cursor. For example, if there's a record in the table with Name containing "SalesRep," the object referenced by oDataObject must have a SalesRep property. The Init method, shown in **Listing 14**, instantiates a DynamicFormRenderEngine object and sets its properties the way we want. Note that it specifies my base classes defined in SFCtrls.vcx; feel free to change this to use your own classes instead.

Listing 14. The Init method of SFDynamicRender.

* Create a Dynamic Form rendering engine object and use our classes as defaults.

```
This.oRenderEngine = newobject('DynamicFormRenderEngine', 'DynamicForm.prg')
with This.oRenderEngine
    .cLabelClass          = 'SFLabel'
    .cLabelClassLib       = 'SFCtrls.vcx'
    .cTextboxClass        = 'SFTextBox'
    .cTextboxClassLib     = 'SFCtrls.vcx'
    .cEditboxClass        = 'SFEditBox'
    .cEditboxClassLib     = 'SFCtrls.vcx'
    .cCommandButtonClass = 'SFCommandButton'
    .cCommandButtonClassLib = 'SFCtrls.vcx'
    .cOptionGroupClass    = 'SFOptionGroup'
    .cOptionGroupClassLib = 'SFCtrls.vcx'
    .cCheckboxClass       = 'SFCheckBox'
    .cCheckboxClassLib    = 'SFCtrls.vcx'
    .cComboboxClass       = 'SFComboBox'
    .cComboboxClassLib    = 'SFCtrls.vcx'
    .cSpinnerClass        = 'SFSpinner'
    .cSpinnerClassLib     = 'SFCtrls.vcx'
```

* Specify that we don't want the container resized to fit the controls, we don't want
* Save and Cancel buttons (we'll use buttons on the form), we're using the form's
* data object, we don't want controls numbered, and we want objects 5 pixels apart
* vertically.

```
    .lResizeContainer     = .F.
    .cFooterMarkup        = ''
    .cDataObjectRef       = 'Thisform.oDataObject'
    .lAddControlNumberToName = .F.
    .nVerticalSpacing     = 5
endwith
```

* Create a collection of behavior objects.

```
This.oBehaviors = newobject('SFCollection', 'SFCtrls.vcx')
```

The Render method (**Listing 15**), called from the form's Init method, creates the markup code Dynamic Form uses to render the controls from the records in the current workarea and has the DynamicFormRenderEngine object render the controls to the specified container (in this example, the form).

Listing 15. The Render method of SFDyanmicRender.

```
#define ccCR chr(13)
lparameters toContainer
local lcRender, ;
    lcName, ;
    loBehavior, ;
    llSetupBehaviors
```

* Create Dynamic Form markup to create a control for each record in the current

* workarea.

```
This.oContainer = toContainer
lcRender       = ''
scan
  lcName       = trim(NAME)
  lcRender     = lcRender + lcName + " .Caption = '" + trim(CAPTION) + "'" + ccCR
  if not empty(CLASS)
    lcRender   = lcRender + ".Class = '" + trim(CLASS) + "'" + ccCR + ;
    iif(empty(LIBRARY), '', ".ClassLibrary = '" + trim(LIBRARY) + "'" + ccCR)
  endif not empty(CLASS)
  if TOP > 0
    lcRender   = lcRender + " .Top = " + transform(TOP) + ccCR
  endif TOP > 0
  if LEFT > 0
    lcRender   = lcRender + " .Left = " + transform(LEFT) + ccCR
  endif LEFT > 0
  if HEIGHT > 0
    lcRender   = lcRender + " .Height = " + transform(HEIGHT) + ccCR
  endif HEIGHT > 0
  if WIDTH > 0
    lcRender   = lcRender + " .Width = " + transform(WIDTH) + ccCR
  endif WIDTH > 0
  if ANCHOR > 0
    lcRender   = lcRender + " .Anchor = " + transform(ANCHOR) + ccCR
  endif ANCHOR > 0
  if not empty(PROPERTIES)
    lcRender   = lcRender + PROPERTIES + ccCR
  endif not empty(PROPERTIES)
  lcRender     = lcRender + '|' + ccCR
```

* If we have behavior code, create a behavior object to handle it.

```
  if not empty(CODE)
    loBehavior = newobject('SFDynamicBehavior', 'SFDynamicForm.vcx')
    loBehavior.cEvents = EVENTS
    loBehavior.cCode   = CODE
    This.oBehaviors.Add(loBehavior, lcName)
    llSetupBehaviors = .T.
  endif not empty(CODE)
endscan
```

* If we have any behavior code, call SetupBehaviorEvents for every control
* created.

```
if llSetupBehaviors
  bindevent(This.oRenderEngine, 'AddControl', This, 'SetupBehaviorEvents', 1)
endif llSetupBehaviors
```

* Render the markup into the container.

```
with This.oRenderEngine
  .oDataObject = This.oDataObject
  .cBodyMarkup = lcRender
  .Render(toContainer)
```

```
endwith

* Clean up before we exit.

if llSetupBehaviors
    unbindevent(This.oRenderEngine)
endif llSetupBehaviors
This.oContainer = .NULL.
```

The behaviors code requires some explanation. Since you can't add code to a class at runtime, if you want a control to do something special in some events, such as Refresh or InteractiveChange, you either need to use a control that has code in those events or you need to use BINDEVENT to bind from the control to an object that handles the events. Assuming you don't want to create special subclasses just for this, I went with the latter. FieldDef.dbf (or whatever table you're using with this class) has an EVENTS field that lists the events we need to bind to and a CODE field that contains the code for the events. The Render method instantiates an SFDynamicBehavior object and fills in its cEvents and cCode properties with the values of those fields, then adds the object to a collection. Once all of the records have been processed, Render uses BINDEVENT to bind the AddControl method of the DynamicFormRenderEngine object, which creates a control, to the SetupBehaviorEvents method (**Listing 16**). This method finds the control the engine just created and for each of the events the behavior object is supposed to bind to, binds that event to the appropriate Handle method of the behavior object.

Listing 16. The SetupBehaviorEvents method of SFDynamicRender.

```
lparameters tcControlClass, ;
    tcClassLib, ;
    tcControlSource, ;
    tcDataType
local loControl, ;
    lcControl, ;
    loBehavior, ;
    laEvents[1], ;
    lnEvents, ;
    lnI, ;
    lcEvent
loControl = This.oContainer.Controls(This.oContainer.ControlCount)
if lower(loControl.BaseClass) <> 'label'
    lcControl = substr(loControl.Name, 4)
    loBehavior = This.oBehaviors.Item(lcControl)
    if not isnull(loBehavior)
        loBehavior.oObject = loControl
        lnEvents = alines(laEvents, loBehavior.cEvents, 5, ',')
        for lnI = 1 to lnEvents
            lcEvent = laEvents[lnI]
            if pemstatus(loControl, lcEvent, 5)
                bindevent(loControl, lcEvent, loBehavior, 'Handle' + lcEvent, 1)
            endif pemstatus(loControl, lcEvent, 5)
        next lnI
    endif not isnull(loBehavior)
endif lower(loControl.BaseClass) <> 'label'
```

Here's an example. One of the custom fields in the sample FieldDef table is the type of company: Prospect or Customer. If they're a Customer, we want to record the name of their salesperson and their yearend. If they're a Prospect, we don't need that information. So, we want the control for Type to refresh the form when its value changes and the controls for salesperson and year end to disable themselves if Type is Prospect when they're refreshed. The Events field in FieldDef for the Type custom field contains "InteractiveChange" and the Code field contains:

```
lparameters toObject, ;
    tcEvent
toObject.Parent.Refresh()
```

Code in the Code field has to accept two parameters: a reference to the control the code applies to and the name of the event in upper case. If a control needs to handle more than one event, use a CASE statement like this:

```
lparameters toObject, ;
    tcEvent
do case
    case tcEvent = 'INTERACTIVECHANGE'
        toObject.Parent.Refresh()
    case tcEvent = 'REFRESH'
        * do something
endcase
```

So, through event binding, when the user changes the value of Type, the form is refreshed.

The Events field for both the SalesRep and YearEnd custom fields contains "Refresh" and the Code field contains:

```
lparameters toObject, ;
    cEvent
toObject.Enabled = toObject.Parent.opgType.Value = 2
```

This causes the controls for those fields to only be enabled when Type is Customer (the second choice in the option group specified as the control for Type).

We won't look at the code in SFDynamicBehavior, as it's quite simple: the various Handle methods simply use EXECSCRIPT to execute the code specified in the cCode property, which comes from the Code field in the definition table.

Main.prg in the code accompanying this document calls the CustomFields class like this:

```
loForm = newobject('CustomFields', 'FieldControls.vcx', '', 'ACME Supplies Ltd.', 1)
loForm.Show()
```

In a real-world application, the form would likely be displayed by clicking a button in the Customers form or choosing a "Edit Custom Fields" menu item and rather than hard-coding the customer name and ID, something like TRIM(CUSTOMERS.COMPANYNAME), CUSTOMERS.CUSTOMERID would be used.

The resulting form is shown in **Figure 23**. The values displayed in the form are stored in CustomFields.dbf. Adding a new custom field is as simple as adding a new record to FieldDef.dbf and a new field to CustomFields.dbf.

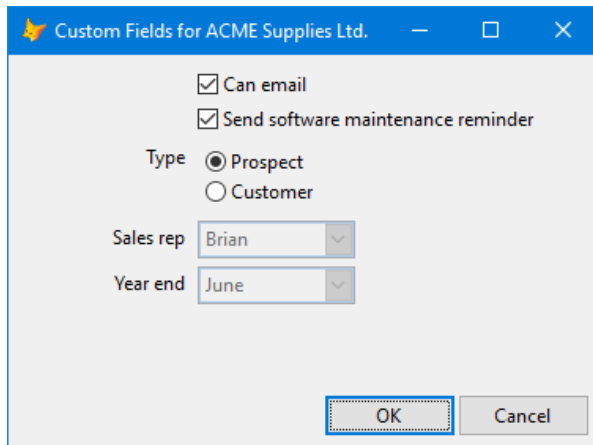


Figure 23. The custom fields form.

Dynamic application settings

Most applications need configuration settings of some type. Configurable settings allow a user to customize how the application works for them. Examples of settings are where data files are located (making that configurable allows the user to run the application on their local workstation but store the data on a server where it can be shared), what email settings to use (if the application sends emails), and what the company's yearend is (if the application works with accounting data).

After creating what seemed like the hundredth dialog allowing a user to change configuration settings, I decided to create a generic one. I've always like the look of the Visual Studio Options dialog (**Figure 24**), so I modeled my form after that.

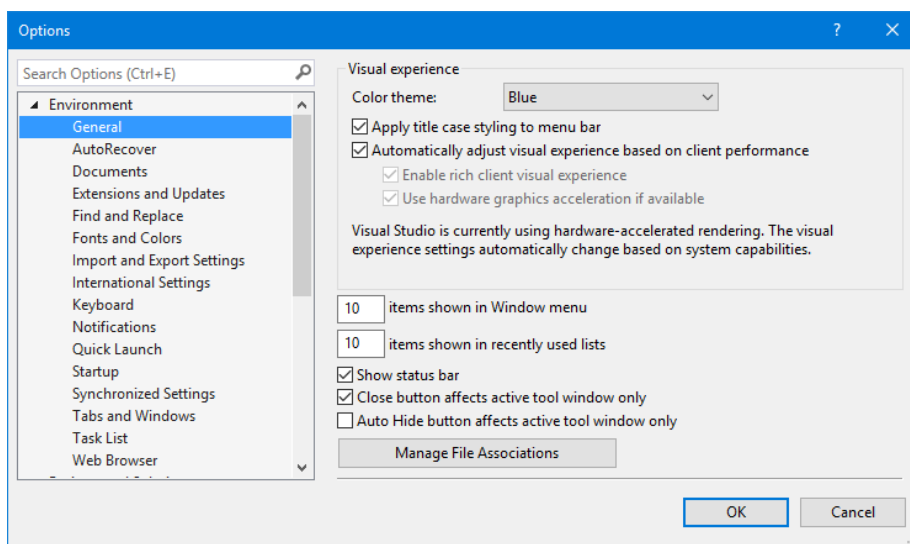


Figure 24. The Visual Studio Options dialog.

Let's start with where the settings are stored. They can be stored in lots of places, but a table, an INI file, and the Windows Registry are the most common. To be flexible, I use a set of persistence classes I wrote about a very long time ago (see "Persistence without Perspiration" at <http://doughennig.com/papers/default.html>) that have the same programmatic interface but different storage mechanisms: SFPersistentINIFile is for INI files, SFPersistentTable is for tables, and SFPersistentRegistry is for the Registry. Here's an example that uses Settings.ini to store settings:

```
loPersist = newobject('SFPersistentINIFile', 'SFPersist.vcx', '', .T.)
loPersist.cFilePath = 'Settings.ini'
loPersist.cSection = 'Options'
```

Here's one that stores settings in the Registry in the specified key under HKEY_CURRENT_USER:

```
loPersist = newobj('SFPersistentRegistry', 'SFPersist.vcx', '', .T.)
loPersist.cKey = 'Software\Stonefield Software Inc.\MyApplication\Options'
```

How does the persistence object know what values to store? That's handled with an options class, SFOptions in SFOptions.vcx. SFOptions uses a data-driven approach; it uses a table whose name is specified in the cOptionsFile property (defaults to Options.dbf) to define which settings are used.

Options.dbf has almost the same structure as FieldDef.dbf that I discussed earlier because it's used by SFDynamicRender to display the settings to the user. It has a few additional fields:

- Rectype contains "P" for a Page record (the item in the TreeView defining a page of settings) or "O" for an Option record (a specific setting).
- Page contains the name of the page the option belongs to.
- Parent contains the name of the parent page the current page belongs under. This allows hierarchical pages. **Figure 25** shows an example of this: Email appears under the General item. To have Email appear at the top level instead, simply blank the Parent field in the sample Options.dbf accompanying this document.
- GetValue contains custom code to execute to convert the value from how it's stored to how it's displayed. For example, GetValue contains the following code for the Password record to decrypt the encrypted password:

```
lparameters toOptions, ;
    tcProperty
set library to VFPEncryption
lcValue = evaluate('toOptions.' + tcProperty)
if not empty(lcValue)
    lcValue = Decrypt(lcValue, 'MyKey', 1024)
endif not empty(lcValue)
return lcValue
```

- SaveValue contains custom code to convert the value from how it's displayed to how it's stored. SaveValue for the Password record contains similar code to GetValue but calls Encrypt instead.

There are two main methods in SFOptions: LoadSettings and SaveSettings. We'll just look at the code for LoadSettings, shown in **Listing 17**.

Listing 17. The LoadSettings method of SFOptions.

```
local lnSelect, ;
    lcAlias, ;
    loException, ;
    lcProperty, ;
    lcDataType, ;
    luValue
```

* Open the options table and create properties for each of the settings, plus add an
* item to the persistent object so it knows what to save and restore.

```
lnSelect = select()
lcAlias = sys(2015)
with This
    select 0
    try
        use (.cOptionsFile) alias (lcAlias) again shared
    catch to loException
        .cErrorMessage = loException.Message
    endtry
    if used(lcAlias)
        scan for RECTYPE = 'O'
            lcProperty = trim(NAME)
            lcDataType = DATATYPE
            do case
                case lcDataType = 'C'
                    luValue = ''
                case lcDataType = 'L'
                    luValue = .F.
                case lcDataType = 'D'
                    luValue = {/}
                case lcDataType = 'T'
                    luValue = {/:}
                otherwise
                    luValue = 0
            endcase
            try
                addproperty(This, lcProperty, luValue)
                .oPersist.AddItem(lcProperty, 'This.oObject.' + lcProperty, lcDataType)
            catch to loException
                .cErrorMessage = loException.Message
            endtry
        endscan for RECTYPE = 'O'
```

* Now use the persistence object to load the values of the settings. Note that to
* avoid dangling references, we don't keep the object reference around any longer

* than we need to.

```
.oPersist.oObject = This
.oPersist.Restore()
.oPersist.oObject = .NULL.
```

* Handle any settings that have code to execute to get the value.

```
scan for RECTYPE = 'O' and not empty(GETVALUE)
  lcProperty = trim(NAME)
  store execscript(GETVALUE, This, lcProperty) to ('This.' + lcProperty)
endscan for RECTYPE = 'O' ...
use
endif used(lcAlias)
endwith
select (lnSelect)
```

This code goes through the options table and adds properties to itself for each one. That allows you to access the settings using code like `loSettings.YearStart` and `loSettings.DataDirectory`. `LoadSettings` also configures the persistence object stored in `oPersist` so it knows which settings to read and write. Then it calls `Restore` to actually load the settings from the appropriate persistence location, and finally executes any custom code in the `GetValue` memo for each record. `SaveSettings` is similar but it calls `Save` rather than `Restore` and executes custom code in the `SaveValue` memo.

To load the settings for an application at startup, do something like this:

```
loPersist = newobject('SFPersistentINIFile', 'SFPersist.vcx', '', .T.)
loPersist.cFilePath = 'Settings.ini'
loPersist.cSection = 'Options'
loPersist.lSaveOnDestroy = .F.
loSettings = newobject('SFOptions', 'SFOptions.vcx', '', loPersist)
loSettings.LoadSettings()
```

Here's an example that uses the `YearStart` setting, which contains which month a company's fiscal year starts in (for example, 11 for November), to determine which fiscal quarter a date is in:

```
lnQuarter = quarter(ldDate, loSettings.YearStart)
```

`SFOptionsDialog` (shown in Figure 25) is the class used to edit the settings. It consists of an `SFTreeViewContainer` object (see "The Mother of All TreeViews" part 1 and 2 at <http://doughennig.com/papers/default.html> for a discussion of that class) to display the list of pages of settings and a pageframe with `Tabs` set to `.F.` to contain the controls to edit the setting. Selecting an item in the `TreeView` sets the `ActivePage` property of the pageframe to the specified page.

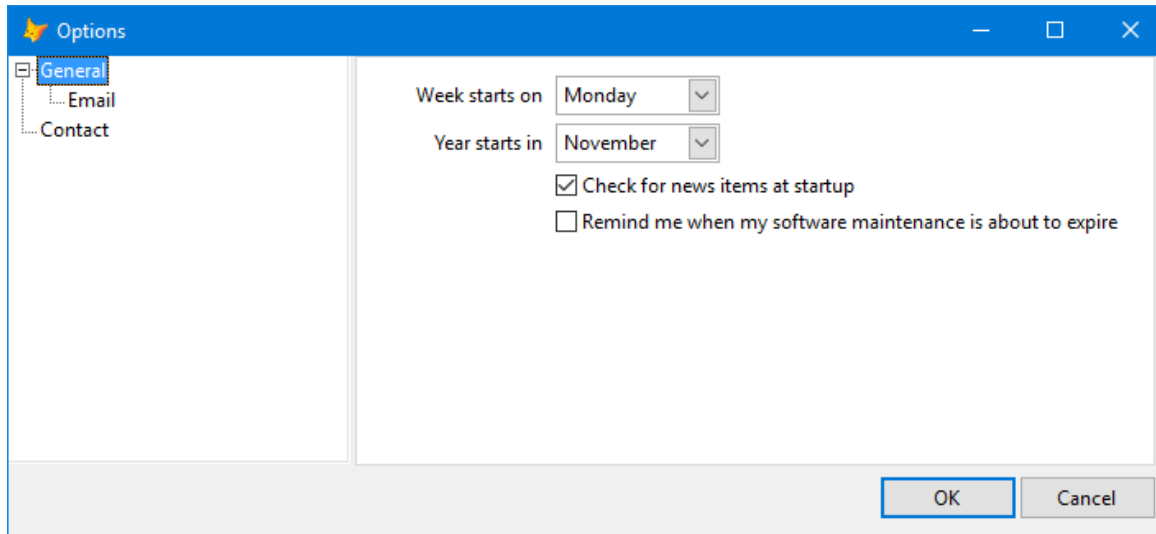


Figure 25. SFOptionsDialog allows the user to edit their settings.

We won't look at the code in this form other than the part related to Dynamic Forms. The following is used to get a list of the options for the current page and render them to the appropriate page of the pageframe:

```
select * from OPTIONS ;
  where RECTYPE = 'O' and trim(PAGE) == lcPage ;
  into cursor RENDER ;
  order by ORDER
loPage = Thisform.pgfoptions.Pages[lnPage]
Thisform.oRender.Render(loPage)
```

As with the CustomFields class we saw earlier, SFOptionsDialog uses an SFDynamicRender object to do the hard work. To display the Options dialog to the user, use code like this:

```
loForm = newobject('SFOptionsDialog', 'SFOptions.vcx', '', loSettings)
loForm.Show()
```

Summary

If you need the ability to create forms that appear differently for different users or different applications, Dynamic Forms is a great tool that'll save you a lot of time. Being able to data-drive the markup code makes Dynamic Forms even more flexible than I originally imagined. Feel free to use the code or the ideas in this document in your own applications.

XLSXWorkbook

Most applications I've worked on had to interact with Microsoft Excel documents in one way or another. Outputting data to Excel is extremely common, as most people are comfortable working with Excel and often want to analyze or massage the data in their applications in ways a developer can't foresee. But importing data from Excel is also popular, both because it can be a common interchange format between different

applications and because people often treat Excel like a database, with its row and column layout making data entry fast and easy.

Built-in commands

The VFP COPY TO and EXPORT TO commands support creating Excel 2.0 (using the “TYPE XLS” clause) and 5.0 (using “TYPE XL5”) files. Newer versions of Excel can open both types of files, but there are some problems:

- Both formats ignore memo fields.
- You may not be able to output all records: VFP 8 and 9 only output a maximum of 65,535 rows, and with VFP 7 and earlier it’s only 16,384.
- Neither command can create an XLSX file (Excel 2007 and later).
- Excel 2.0 format doesn’t handle date fields correctly: they give a “text date with 2-digit year” warning and have to be edited to work as actual date values.

The first and second points are usually the biggest problems for most people. If you can live with these issues, then these commands are your best solution since they’re fast and take only one line of code.

The VFP APPEND FROM and IMPORT FROM commands support reading from Excel version 2.0 (using the “TYPE XLS” clause), version 5.0 (using “TYPE XL5”), and Excel 97 (using “TYPE XL8”) files (all versions are XLS files). APPEND FROM appends records into an existing table while IMPORT FROM creates a new table, either a free table or one in a database container, depending on keywords you supply with the command (see VFP help for details). As with exporting data, there are a few issues:

- Reading from some XLS files saved from Excel 2007 or later causes errors; for example, importing from EmployeesFromXLSX.xls included with the sample files for this document causes VFP to crash.
- The columns in the table created with IMPORT FROM are named A, B, C, etc. You have to use ALTER TABLE or MODIFY STRUCTURE to give them more meaningful names.
- Column headings in the first row are treated as a record so you have to delete that record after reading from the Excel file.
- Memo fields are ignored. For example, USE EMPLOYEES to open Employees.DBF (included with the sample files for this document), then APPEND FROM Employees.XLS TYPE XLS. The last column in Employees.XLS is a 254-character text field but after the APPEND FROM command, you’ll find the NOTES memo field, the last field in the table, is empty.
- There is no support for reading from XLSX files (Excel 2007 and later).

Again, if you can live with these issues, especially the last one, then these commands are your best solution since they're the fastest way to import from Excel and take only one line of code.

XLSXWorkbook

If the export and import issues are deal-breakers, you have other choices. Some of them need COM and require that Excel be installed, which can be an issue on servers or when called from a scheduled task. Greg Green's XLSXWorkbook VFPX project, on the other hand, writes directly to XLSX files, which are actually XML files zipped into a file with an XLSX extension, so Excel isn't required. You can download it from <https://github.com/ggreen86/XLXS-Workbook-Class>; the download includes a number of test programs and forms, but all of the actual code for the utility is in a single VCX, VFPxWorkbookXLSX.VCX. The download also includes extensive documentation in a PDF file.

To use XLSXWorkbook, start by instantiating the VFPXWorkbookXLSX class in VFPXWorkbookXLSX.VCX. XLSXWorkbook has a couple of methods that make quick work of creating a spreadsheet from VFP data: SaveTableToWorkbook, which saves the specified table to a workbook, and SaveGridToWorkbook, which save the content and some of the formatting of the specified grid to a workbook.

The code in **Listing 20** outputs the Employee sample table to EmployeeXLSXWorkbook.XLSX. This file is very similar to those created using the other techniques.

Listing 18. Test code for XLSXWorkbook.

```
open database (_samples + 'data\testdata')
use Employee
loExport = newobject('VFPXWorkbookXLSX', 'VFPXWorkbookXLSX\VFPXWorkbookXLSX.vcx')
loExport.SaveTableToWorkbook(alias(), 'EmployeeXLSXWorkbook.xlsx', .T., .T.)
```

Creating a workbook from a grid is easy too, needing just a single line of code:

```
loExport.SaveGridToWorkbook(This.grdGrid, 'MyWorkbook.xlsx', .T., .T.)
```

However, XLSXWorkbook provides a lot more control over the workbook if you wish. You can format cells (color, font, border, style, etc.), manage worksheets within the workbook, use custom numeric formatting, and so on. So, you don't have to create a plain spreadsheet if your users want something more attractive. The code in FormatXLSXWorkbook.PRG, shown in **Listing 19** and included with the code accompanying this document, creates the workbook shown in **Figure 26**.

	A	B	C	D	E
1	4NW0UDLNE	_4NW0UDLNG	_4NW0UDLNH	_4NW0UDLNI	_4NW0UDLNJ
2	4NW0UDLNO	_4NW0UDLNP	_4NW0UDLNQ	4NW0UDLNR	_4NW0UDLNS
3	4NW0UDLNK	_4NW0UDLNY	_4NW0UDLNZ	4NW0UDLOO	_4NW0UDLOP
4	4NW0UDLO6	_4NW0UDLO7	_4NW0UDLO8	_4NW0UDLO9	_4NW0UDLOA
5	_4NW0UDLOF	4NW0UDLOG	_4NW0UDLOH	_4NW0UDLOI	_4NW0UDLOJ
6	_4NW0UDLOO	_4NW0UDLOP	_4NW0UDLOQ	_4NW0UDLOR	_4NW0UDLOS
7	4NW0UDLOX	_4NW0UDLOY	_4NW0UDLOZ	_4NW0UDLP0	_4NW0UDLP1
8	4NW0UDLP6	_4NW0UDLP7	_4NW0UDLP8	_4NW0UDLP9	_4NW0UDLPA
9	_4NW0UDLPF	_4NW0UDLPG	_4NW0UDLPH	_4NW0UDLPI	_4NW0UDLPJ
10	_4NW0UDLPO	_4NW0UDLPP	_4NW0UDLPQ	_4NW0UDLPR	_4NW0UDLPS
11					

Figure 26. This formatted Excel workbook was created with XLSXWorkbook.

Listing 19. This code creates a formatted Excel document.

```
#DEFINE BORDER_LEFT           1
#DEFINE BORDER_RIGHT         2
#DEFINE BORDER_TOP           4
#DEFINE BORDER_BOTTOM        8

loExport = newobject('VFPXWorkbookXLSX', 'VFPXWorkbookXLSX\VFPXWorkbookXLSX.vcx')
with loExport
    lnWorkbook = .CreateWorkbook('ExcelTest.xlsx')
    lnSheet    = .AddSheet(lnWorkbook, 'Test Sheet 1')

* Create some dummy data.

    for lnRow = 1 to 10
        for lnCol = 1 to 9
            .SetCellValue(lnWorkbook, lnSheet, lnRow, lnCol, sys(2015))
        next lnCol
    next lnRow

* Set the row and column heights.

    .SetRowHeight(lnWorkbook, lnSheet, 6, 25)
    .SetColumnWidth(lnWorkbook, lnSheet, 1, 25)
    for lnCol = 2 to 9
        .SetColumnWidth(lnWorkbook, lnSheet, lnCol, 14)
    next lnCol

* Set font, size, color, and style.

    .SetCellFont(lnWorkbook, lnSheet, 1, 1, 'Calibri', 14, .T., .T., rgb(255, 0, 0))
    .SetCellFont(lnWorkbook, lnSheet, 2, 1, 'Tahoma', , , , rgb(0, 0, 255))
    .SetCellFont(lnWorkbook, lnSheet, 3, 1, , 14, .T.)
    .SetCellFont(lnWorkbook, lnSheet, 4, 1, 'Arial', 14, .T., .T., rgb(0, 0, 255))
    .SetCellFont(lnWorkbook, lnSheet, 6, 1, , , , , , 'single')
    .SetCellFont(lnWorkbook, lnSheet, 7, 1, , , , , , 'double')

* Set borders.

    lnBorder = BORDER_LEFT + BORDER_RIGHT + BORDER_TOP + BORDER_BOTTOM
    .SetCellBorder(lnWorkbook, lnSheet, 3, 4, lnBorder, 'thin', rgb(16, 100, 200))
    .SetCellBorder(lnWorkbook, lnSheet, 3, 6, lnBorder, 'thick', rgb(100, 150, 200))
    .SetCellBorder(lnWorkbook, lnSheet, 3, 8, lnBorder, 'double', rgb(200, 150, 100))
```

```
.SetCellBorderEx(lnWorkbook, lnSheet, 5, 2, 'thin', , 'thin', , 'thick', , 'thick')
```

* Save and open the workbook.

```
.SaveWorkbook(lnWorkbook)  
endwith  
Execute('ExcelTest.xlsx')
```

XLSXWorkbook can also import data from Excel. It doesn't have a specific method to read all the rows from a worksheet into a table, but it does have a method, `GetSheetRowValues`, that reads all columns from the specified row in the specified worksheet, so you just need a loop to process all rows. Pass `GetSheetRowValues` a handle to a document you opened with `OpenXlsWorkbook`, the name or number of a worksheet in the document, and the row number to read from. `GetSheetRowValues` returns an object with a `Count` property, containing the number of columns read, and a `Values` array, with one row per column read. The first column in the array contains the cell value and the second column the data type of the value.

The code in **Listing 20** imports `Employees.XLSX` into `Employees.DBF`.

Listing 20. Test code for XLSXWorkbook.

```
use Employees exclusive  
zap  
loExport = newobject('VFPXWorkbookXLSX', 'VFPXWorkbookXLSX\VFPXWorkbookXLSX.vcx')  
lnWB     = loExport.OpenXlsWorkbook('Employees.xlsx')  
llDone   = .F.  
lnRow    = 2  
do while not llDone  
    loRow = loExport.GetSheetRowValues(lnWB, 1, lnRow)  
    lnRow = lnRow + 1  
    if not isnull(loRow) and not isnull(loRow.Values[1, 1])  
        append blank  
        for lnI = 1 to loRow.Count  
            lcField = field(lnI)  
            replace &lcField with loRow.Values[lnI, 1]  
        next lnI  
    else  
        llDone = .T.  
    endif not isnull(loRow) ...  
enddo while not llDone  
browse
```

Summary

XLSXWorkbook is a great addition to the VFPX family. I use it in Stonefield Query, for both outputting a report and documenting the data dictionary. It can both read from and write to formatted Excel documents. I highly recommend this project.

FoxBin2PRG

FoxBin2PRG is one of several utilities that convert FoxPro binary files, such as SCXs, VCXs, FRXs, and so on, to text files so they can be compared to other versions of those files (such as a previous version in a version control system). FoxBin2PRG has numerous advantages over its competitors including:

- Its author, Fernando Bozzo, continues to enhance and support it and is very responsive to issues (often less than 24 hours response time).
- It provides two-way conversion so it can be used to regenerate the binary files after merging changes made by other developers or on different branches in the text equivalents.
- The text files it generates look like PRGs (although they cannot be compiled or run), making them easily readable to a VFP developer.
- The methods and properties of classes and forms are sorted alphabetically for easy comparison.
- It can process a single file or all the files in a project.
- It's highly configurable.

You can find FoxBin2PRG at <https://github.com/fdbozzo/foxbin2prg>.

I'm not going to go into a lot of detail on FoxBin2PRG; it is well documented in its GitHub repository. I'm going to focus on how to use it and how to configure it.

Using FoxBin2PRG

There are several ways you can use FoxBin2PRG:

- If you create the appropriate shortcuts in the Windows Send To folder (the documentation describes how), you can right-click a VFP binary file and convert it to text or vice versa. Although this is a manual task, it's very convenient for processing a single file.
- You can run FoxBin2PRG.exe, passing it the appropriate parameters, such as the name of a VFP binary file to convert to text. This can be automated to process multiple files, including from a BAT file or PowerShell script.
- You can run FoxBin2PRG.prg, passing it the appropriate parameters, such as the name of a VFP binary file to convert to text. Note that the VFP path has to be set to include the folder where FoxBin2PRG is installed so it can find its support files.
- You can instantiate the `c_foxbin2prg` class in FoxBin2PRG.prg and call the desired methods. This can be automated from VFP code to process multiple files.

Here are some of the ways I use it:

- After adding, deleting, or edit records in a VFP table that's built into an application (such as meta data), I right-click the DBF file and choose "FoxBin2PRG – Binary to Text" from the Send To submenu in the shortcut menu, since it's typically the text file I keep in version control not the binary file.
- After editing the help file for an application using West Wind's HTML Help Builder, I copy the HBP file (the source for the CHM, which is really a VFP table with a different extension) to a DBF file (since FoxBin2PRG doesn't know the HBP file is a table) then use FoxBin2PRG to generate the text equivalent by running a BAT file:

```
copy sqconfig.hbp sqconfig.dbf
path\foxbin2prg sqconfig.dbf
```

- Project Explorer uses a little more complex version of the following code to convert binary to text and vice versa (simplified here for readability):

```
loConvert = newobject('c_foxbin2prg', This.cConverterPath + ;
    'foxbin2prg.prg')
```

```
* Get the FoxBin2PRG settings for the folder the file is in.
```

```
loSettings = loConvert.get_DirSettings(justpath(tcFile))
```

```
* If the file type is supported, convert it. If it's a DBF, close it first.
```

```
if not empty(This.GetFileType(tcFile, loSettings))
    if lower(justext(tcFile)) = 'dbf'
        CloseFile(tcFile)
    endif lower(justext(tcFile)) = 'dbf'
    loConvert.Execute(tcFile, iif(tlAll, '*', ''))
endif not empty(This.GetFileType(tcFile, loSettings))
```

Since you can configure FoxBin2PRG to not convert certain types of VFP binaries, this code calls the GetFileType method that checks whether we should convert the specified file. The code in that method includes the following and returns an empty string if the specified file shouldn't be processed:

```
lcExt    = lower(justext(tcFile))
llBinary = inlist(lcExt, 'pjx', 'vcx', 'scx', 'mnx', 'frx', 'lbx', 'dbf', ;
    'dbc')
do case
case lcExt = 'dbf' and toSettings.DBF_Conversion_Support > 0
    lcType = lower(toSettings.c_DB2)
case lcExt = 'dbc' and toSettings.DBC_Conversion_Support > 0
    lcType = lower(toSettings.c_DC2)
case llBinary and evaluate('toSettings.' + lcExt + ;
    '_Conversion_Support') > 0
    lcType = lower(evaluate('toSettings.c_' + strtran(lcExt, 'x', '2')))
```

Configuring FoxBin2PRG

FoxBin2PRG's behavior can be configured using a file named FoxBin2PRG.cfg, which is just a text file containing settings. A template for that file named FoxBin2PRG.cfg.txt is included

with FoxBin2PRG. If you want to change any of the default settings, copy the file, rename it to FoxBin2PRG.cfg, make the desired changes, and put it into one of two places:

- The FoxBin2PRG install folder for global settings (that is, affecting all conversions)
- The source folder for your project for project-specific settings

FoxBin2PRG uses a hierarchy of settings. It uses its default values, then overrides those with any specified in the cfg file in its own folder, then overrides those with any specified in the cfg file in the folder where the binary or text files being processed are located. **Table 3** shows the settings.

Table 3. FoxBin2PRG configuration settings.

Setting	Default	Purpose
Extension	extension: pj2=pj2 extension: vc2=vc2 extension: sc2=sc2 extension: fr2=fr2 extension: lb2=lb2 extension: mn2=mn2 extension: db2=db2 extension: dc2=dc2	Specifies the extension for the text file for each file type. Generally, the text extension replaces the “x” in the binary extension with “2” (for example, PJ2 for a PJX file) except DBF is DB2 and DBC is DC2.
ShowProgressbar	1	Display a progress bar during file processing: 0=don’t show, 1=always show, 2= show only for multi-file processing
DontShowErrors	0	Set to 1 to disable displaying error messages
NoTimestamps	1	Clear timestamps to minimize differences; disable by setting to 0
Debug	0	Set to 1 to create individual <file>.Log files
ExtraBackupLevels	1	The number of BAK files to create/preserve
ClearUniqueID	1	Clear UniqueID values (useful for diff and merge); disable by setting to 0
OptimizeByFilestamp	0	Optimize file regeneration depending on file timestamp: 0=not optimized, 1=skip processing if output file is newer, 2=skip processing if output file timestamp equals input file’s
RemoveNullCharsFromCode	1	Remove NULLs from source code; disable by setting to 0
RemoveZOrderSetFromProps	0	Set to 1 to remove ZOrderSet property from objects
Language	(auto)	Language of messages and lo files: EN=English, FR=French, ES=Español, DE=German
ClearDBFLastUpdate	1	Clear DBF LastUpdate (useful for diff); disable by setting to 0
ExcludeDBFAutoincNextval	0	Set to 1 to exclude this value from db2
PJX_Conversion_Support	2	0=no support, 1=generate text only (useful for diff), 2=generate text and binary (useful for diff and merge)
VCX_Conversion_Support	2	0=no support, 1=generate text only (useful for diff), 2=generate text and binary (useful for diff and merge)

Setting	Default	Purpose
SCX_Conversion_Support	2	0=no support, 1=generate text only (useful for diff), 2=generate text and binary (useful for diff and merge)
FRX_Conversion_Support	2	0=no support, 1=generate text only (useful for diff), 2=generate text and binary (useful for diff and merge)
LBX_Conversion_Support	2	0=no support, 1=generate text only (useful for diff), 2=generate text and binary (useful for diff and merge)
MNX_Conversion_Support	2	0=no support, 1=generate text only (useful for diff), 2=generate text and binary (useful for diff and merge)
DBC_Conversion_Support	2	0=no support, 1=generate text only (useful for diff), 2=generate text and binary (useful for diff and merge)
DBF_Conversion_Support	1	0=no support, 1=generate text for structure only (useful for diff), 2=generate text and binary for structure only (useful for diff and merge), 4=generate text for structure and data (useful for diff), 8=generate text and binary for structure and data (useful for diff and merge)
DBF_Conversion_Included	*	If DBF_Conversion_Support is 4, you can specify filemasks for tables to include in the process
DBF_Conversion_Excluded		If DBF_Conversion_Support is 4, you can specify filemasks for tables to exclude in the process
UseClassPerFile	0	0=one text file for VCX/DBC, 1=multiple file.class.tx2 files, 2=multiple file.baseclass.class.tx2 files including DBC members
RedirectClassPerFileToMain	0	0=don't redirect to file.tx2, 1=redirect to file.tx2 when selecting file.class.tx2
ClassPerFileCheck	0	0=don't check file.class.tx2 inclusion, 1=check file.class.tx2 inclusion

Here's what my FoxBin2PRG.cfg contains:

```
ExtraBackupLevels: 0
DBF_Conversion_Support: 8
```

Summary

I use FoxBin2PRG multiple times a day, both through Project Explorer integration (saving changes to a form, class, report, or label automatically calls FoxBin2PRG to regenerate the text file) and using the Send To shortcuts to manually generate text from binary or vice versa. If you use version control, FoxBin2PRG is the tool you need to make diffs and merges possible.

Summary

VFPX has changed a lot in the past few years. It has a completely new home, is much more oriented towards version control and collaboration, and has several new useful projects

and enhancements to existing ones. If you're new to VFPX or haven't visited in a while, check it out at <http://vfp.org> and see what projects you can make use of. Of course, we welcome ideas you have for new projects as well.

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*. He wrote over 100 articles in 10 years for FoxRockX and FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe.

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (<http://www.swfox.net>). He is one of the administrators for the VFPX VFP community extensions Web site (<http://vfp.org>). 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>).

