Cool Tools from Craig Boyd

Doug Hennig

Craig Boyd has produced some very cool tools for VFP developers. This month, Doug Hennig examines Craig's encryption library and splitter and progress bar controls.

One of the brightest lights in the VFP constellation is new Microsoft MVP Craig Boyd. He has a very interesting blog (http://www.sweetpotatosoftware.com/SPSBlog/), was the originator of and is one of the administrators of SednaX (http://www.gotdotnet.com/codegallery/codegallery.aspx?id=0826d7a6-1dab-4a71-8e70-f2170c3c1661), a community site dedicated to extending VFP, and has released a lot of very cool code for VFP developers. I am constantly amazed by not only the amount of code he's produced and generously released to the public, but also the ingenious way in which he's implemented his ideas. This article, which marks my tenth anniversary as a FoxTalk author, starts a new series that looks at some of the tools Craig has produced.

Encryption library

In my January 2006 FoxTalk article, "Role-Based Security, Part II," I briefly discussed Craig's VFPEncryption.FLL, a library providing encryption and decryption functions to VFP developers. This month, we'll look at it a little more closely. You can download this library from http://www.sweetpotatosoftware.com/files/vfpencryption.zip.

VFPEncryption.FLL is simple to use: SET LIBRARY TO VFPEncryption.FLL and call one of the five functions:

- Encrypt(cString, cKey[, nType[, nMode]]): encrypts the specified string. I'll discuss the various parameters later.
- Decrypt(cString, cKey[, nType[, nMode]]): decrypts the specified string.
- EncryptFile(cSourceFileName, cTargetFileName, cKey[, nType[, nMode]]): encrypts the specified file, creating the target file as a result (specify full paths for both names).
- DecryptFile(cSourceFileName, cTargetFileName, cKey[, nType[, nMode]]): decrypts the specified file into the target file.
- Hash(cString[, nHashType]): hashes the specified string. A hash is a one-way cipher; there's no way to decrypt it. This is often used for passwords; you store the initial password as a hash, then when a user logs in, hash the password they enter and compare that to the stored hash. The beauty of a hashed value is that, unlike other mechanisms, there's no private key that could be compromised.

Here's a description of the parameters:

- cKey: the key used for encryption and decryption. Obviously, you must provide the same key you used to encrypt a string or file when you decrypt it. The key may need to be of a particular length depending upon the encryption type.
- nType: the type of encryption. VFPEncryption.FLL supports six types of encryption. Specify 0 for AES128, which requires a 16-character key, 1 for AES192, which requires a 24-character key, 2 (the default) for AES256, which requires a 32-character key, 4 for Blowfish, which requires a 56-character key, 8 for TEA, which requires a 16-character key, or 1024 for RC4, in which case the key can be any length.
- nMode: the mode for encryption. Specify 0 (the default) for Electronic Code Book (ECB), 1 for Cipher Block Chaining (CBC), and 2 for Cipher Feedback Block (CFB). This value is ignored if you use RC4 encryption.

- nHashType: the type of hash function to use. Specify 1 for SHA1 (also known as SHA160), 2 for SHA256, 3 for SHA384, 4 (the default) for SHA512, 5 for MD5, 6 for RIPEMD128, or 7 for RIPEMD256.

Don't worry if you don't understand the different types or modes of encryption; I really don't have a clue about this stuff, which is the main reason I like VFPEncryption.FLL! TestVFPEncryption.PRG is a short program that demonstrates the use of this library.

```
set library to VFPEncryption.FLL

* Test encrypting and decrypting a string.

lcString    = 'This is pretty cool stuff!'
lcKey       = padr('This is a secret key phrase.', 32)
  && padded to the length needed by the default
  && encryption type
lcEncrypted = Encrypt(lcString, lcKey)
wait window lcEncrypted
wait window Decrypt(lcEncrypted, lcKey)

* Test encrypting and decrypting a file.

EncryptFile(fullpath('TestVFPEncryption.prg'), ;
  fullpath('EncryptedFile.prg'), lcKey)
modify file EncryptedFile.prg
DecryptFile(fullpath('EncryptedFile.prg'), ;
  fullpath('DecryptedFile.prg'), lcKey)
modify file DecryptedFile.prg

* Test hashing a string.

wait window Hash(lcString)
```

Splitter control

Splitters are interesting controls: they don't really have a visual appearance themselves, but they allow you to change the relative size between two or more other controls by adjusting the size of one at the expense of the other. Splitters appear in lots of places in Windows applications; for example, in Windows Explorer, you can adjust the relative sizes of the left and right panes using a splitter. Splitters can be horizontal (they adjust objects to the left and right) or vertical (they adjust objects above and below), and you could have both types of splitter on the same form.

In my July 1999 FoxTalk column, "Splitting Up is Hard to Do," I presented a VFP splitter class, SFSplitter, and specific subclasses for horizontal (SFSplitterH) and vertical (SFSplitterV) splitters. They were fairly complex classes: they automatically added another class, SFSplitterCover, to their parent and used OLE drag and drop to implement splitter movement. Adding a splitter to a form required dropping the appropriate splitter class on the form and setting some properties that indicated the names of the controls affected by moving the splitter. While the classes were quite difficult to create, I've used them successfully in many projects and don't have to worry about the underlying complexity.

Last summer, Craig posted a different implementation of a splitter class on his blog, and I must admit I'm jealous. There's only a single class, Splitter in Splitter.VCX, with just one property to set—Vertical, which indicates whether the splitter is vertical or horizontal—and not very much code. It assumes every visual control to the left and right (in the case of a vertical splitter) or above and below (in the case of a horizontal one) the splitter are affected when the splitter moves, a reasonable assumption, although you can place the string "DoN't_MoVe_SpLiT" into the Tag of any control that should be ignored.

Let's see how it works, then delve into the code. Download http://www.sweetpotatosoftware.com/files/splitter.zip from Craig's Web site, unzip it into a directory, and DO FORM Splitter. This sample form, shown in **Figure 1**, has three edit boxes and two splitters, one horizontal and the other vertical. Move the horizontal splitter up and down and notice all three edit boxes are sized appropriately. Move the vertical splitter left and right; the two lower edit boxes are adjusted as necessary.

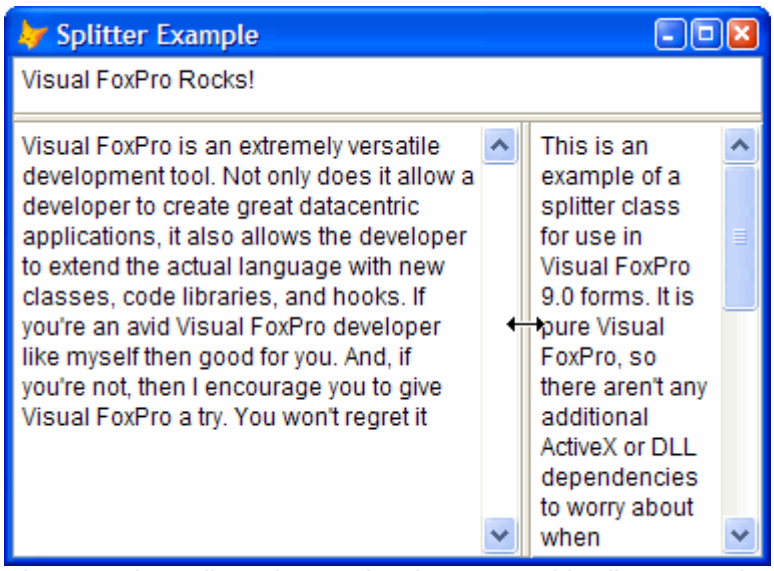

Figure 1. The Splitter class makes it easy to add splitter controls to your forms.

Now modify the form and notice that other than being sized and positioned appropriately, the only custom changes to the properties of the two splitter controls are Anchor (so they respond to form resizing) and Vertical (indicating how the splitter behaves). Let's look at the Splitter class to see how it works.

The MousePointer property is set to 9-Size WE, but the Init method changes it to 7-Size NS if Vertical is .F. Thus, moving the mouse over the control changes the pointer to the appropriate shape, making it obvious to the user that this is a splitter control. The MouseDown method saves the current X or Y coordinate of the mouse (depending on the setting of Vertical) to the custom MouseDownAt property. MouseUp and MouseLeave both set this property to 0.

The actual work is done in the MouseMove and Move methods. I won't show the code in these methods for space reasons, but will describe them in detail.

MouseMove, which fires when the mouse is moved over the control, ensures the left mouse button is pressed and MouseDownAt is greater than 0; this prevents anything from happening if the mouse was first pressed down over another control and then moved over the splitter. If the mouse was moved (its current X or Y coordinate doesn't match MouseDownAt), the code calculates how much it was moved and, as long as it isn't too close to the edge of the form, as determined by the custom MinimumSize property, MouseMove calls the Move method to do the actual movement and reset MouseDownAt to the new position. MinimumSize defaults to 40 so controls don't get so small you can't see or use them, but you could set it to a different value as necessary.

Move does the dirty work of moving any controls that should be affected by the splitter and the splitter itself. It starts by ensuring MouseDownAt isn't 0. It then locks the screen so object movement doesn't appear until they're all done, determines how far the mouse was moved, and goes through the controls in its parent container. Any control that has its Tag property set to "DoN't_MoVe_SpLiT" is ignored, as are non-visual controls. The code saves the value of the Anchor property of the control and sets it to 0; this is necessary or the control won't behave properly the next time the form is resized. If the splitter is vertical and the control is to the left or right of the splitter, the code adjusts the width of the control, as well as its Left property if it's to the right of the splitter. If the splitter is horizontal, the code does something similar to controls above and below it. The Anchor property is then restored. After processing all controls, the Anchor property of the splitter is saved, the splitter is moved to its new position, and Anchor is restored. Finally, the screen is unlocked.

If you want to programmatically move the splitter, set MouseDownAt to a non-zero value and call Move. This can be used, for example, to restore the splitter to the position it was at the last time the user ran the form. For example, you might use code like the following in the Init of the form, where lnTop is the value to use for Top property for a horizontal splitter.

```
with This.Splitter
```

```
    .MouseDownAt = .Top
    .Move(.Left, lnTop, .Width, .Height)
endwith
```

That's it. As I said earlier, there isn't a lot of code in this class, it's much simpler than the splitter class I presented a few years ago, and it's easier to use. However, being the picky person I am, I decided to subclass Splitter to change one behavior: Splitter assumes the minimum size for controls on both sides of it is the same. However, there may be times when one side must be bigger than the other, such as when you have a listbox on one side that could be sized quite narrow but a container of labels and textboxes on the other that shouldn't be.

SFSplitter, in SFSplitter.VCX, is a subclass of Splitter with two new properties: nMinimumSize1, which contains the minimum size for controls to the left or above the splitter, and nMinimumSize2, which contains the minimum size for controls to the right or below the splitter. Init sets them to the appropriate values if you leave them at their defaults of 0. I then overrode MouseMove to use these properties rather than MinimumSize. To see how my subclass works, DO FORM SFSplitter. Notice the splitters aren't symmetrical in their movements; the controls on one size can be bigger than the other because I set nMinimumSize1 and nMinimumSize2 to different values. This form also changes the initial position of the horizontal splitter in Init to show how that works.

Progress bar control

Progress bar controls, sometimes called thermometer controls, provide feedback to a user when a lengthy process runs. Visual FoxPro has long included an ActiveX control, the Microsoft ProgressBar control, you can add to a form to provide this type of feedback. However, one of the downsides of this control is that you have to distribute and register MSCOMCTL.OCX. Also, since it's an old control, it doesn't have a modern appearance like the typical progress bars you see in other applications.

Only a week after posting his splitter control, Craig released his progress bar class, ProgressBar in ProgressBarEx.VCX. It's a 100% VFP code control, so there are no ActiveX issues to worry about. As you can see in **Figure 2**, it looks like a standard Windows XP progress bar, complete with colorful gradients and an optional percentage label.

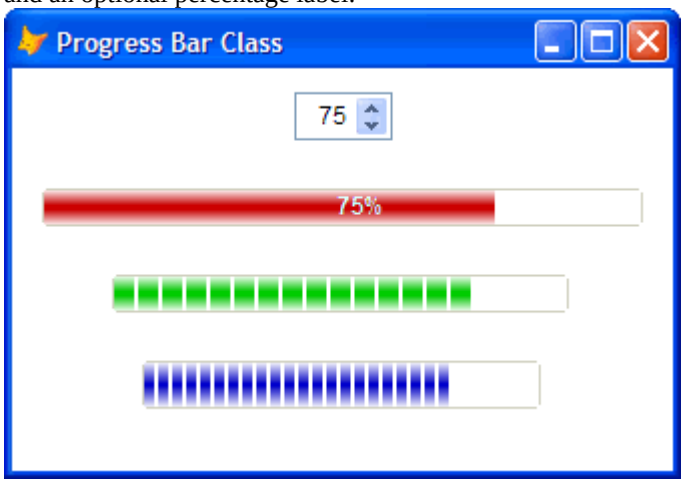

Figure 2. The ProgressBar class provides visual feedback to users about long-running processes.

To obtain Craig's progress bar control, download http://www.sweetpotatosoftware.com/files/progressbarex.zip. To see how it works, run the included Example form and enter a percentage value in the spinner. The three progress bars show the different ways the control can appear.

Let's see how Craig implemented his progress bar. There are two classes in ProgressBarEx.VCX: PBBar, which represents a single segment of the bar and ProgressBar, the progress bar class itself.

ProgressBar has several properties that determine its appearance:

- BarColor: the color of the progress bar. The choices are 1 for red, 2 (the default) for green, and 3 for blue.

- Bars: the number of segments in the progress bar. The higher this number, the narrower each segment is.
- Min: the minimum value for the progress bar; the default is 0.
- Max: the maximum value for the progress bar; the default is 100.
- Percentage: the current percentage complete.
- ShowPercentage: .T. to display the percentage in the progress bar.
- SolidBar: .T. to display a solid bar or .F. (the default) to display the bar as individual segments.
- Value: the current value of the progress bar. You don't have to specify a percentage value. For example, if your code is processing 1,249 records, set Max to 1,249 and Value to the current record number being processed.

The Init method of ProgressBar instantiates the number of instances of PBBar specified in the Bars property, but doesn't set their Visible property to .T. so they initially don't appear. It also adds a label to the control if ShowPercentage is .T. The Value property has an Assign method that determines how many segments to show, and sets the Visible property of those segment objects to .T. It also updates the caption of the label if ShowPercentage is .T. The Percentage property has an Access method that uses the values of Min, Max, and Value to determine the percentage complete.

The actual magic of making the progress bar look like a Windows XP control is in the PBBar class. Its Init method accepts the color, width, and height of each segment (passed by ProgressBar when it instantiates these objects). To display a segment, Init instantiates a set of Line objects, one for each pixel of height in the bar. These lines are drawn with varying colors, from the darkest shade in the middle to lighter shades at the top and bottom, to create a gradient effect. Also, PBBar sets the DrawMode property of the line to 14-Merge Pen Not. This allows the percentage label to appear in the correct color, even when it partially or completely covered by segments.

To use Craig's ProgressBar class, simply drop it on a form, set the properties as desired, and in a processing loop, set Value to the appropriate value. However, rather than using a separate form to display the progress of something, you may want to display the progress in the same form as the process was started from. For example, I like using a status bar in some types of forms and it would be nice to show the progress bar in one of the panels of the status bar. I use Rick Strahl's wwStatusBar control rather than the ActiveX Microsoft StatusBar Control for the same reasons Craig's progress bar is better than the ActiveX control; see http://www.west-wind.com/presentations/wwstatusbar/wwstatusbar.asp for an article describing this class and a link to download the source code.

One slight issue with ProgressBar is that it expects to do all of its setup in Init. The problem with that approach is it expects the properties were set to the desired values in the Properties window. If you instantiate the class programmatically, Init fires before you have a chance to set the properties. So, I created a subclass of ProgressBar called SFProgressBar in SFProgressBar.VCX. I copied the code in ProgressBar.Init to a new method called SetupProgressBar and put a comment into Init so nothing happens when the class instantiates. Because I'm lazy and would rather avoid having to manually call SetupProgressBar, I added a custom property called lSetup, which defaults to .F., and set it to .T. at the end of SetupProgressBar. I then overrode Value_Assign with the following code:

```
lparameters tuNewVal

* Set up the progress bar if it hasn't been done.

if not This.lSetup
  This.SetupProgressBar()
endif not This.lSetup

* If the value is 0, let's be invisible. Otherwise,
* ensure we can be seen.

This.Visible = tuNewVal <> 0
dodefault(tuNewVal)
```

This ensures the progress bar is setup properly the first time Value gets set. Also, I decided the progress bar should be invisible if Value is 0 and visible otherwise, so this code handles that.

I also created a subclass of wwStatusBar called SFStatusBar in SFStatusBar.VCX. The Init method of this class instantiates SFProgressBar into the ProgressBar property, adds a custom Panel to it indicating which panel the progress bar should appear in, and used BINDEVENT() so the new PutProgressBarInPanel method fires when this property is changed. I also added a call to Resize to overcome an issue with wwStatusBar: it doesn't display properly when a form first displays because its Resize method, which does the set up, is bound to the Resize event of the form. If the form isn't resized, the status bar isn't sized properly.

```
This.NewObject('ProgressBar', 'SFProgressBar', ;
  'SFProgressBar.vcx')
addproperty(This.ProgressBar, 'Panel', 1)
bindevent(This.ProgressBar, 'Panel', This, ;
  'PutProgressBarInPanel', 1)
dodefault()
This.Resize()
```

To see SFStatusBar in action, run the SFProgress form and click the Start Process button. As shown in **Figure 3**, the progress bar appears in the second panel of the status bar.

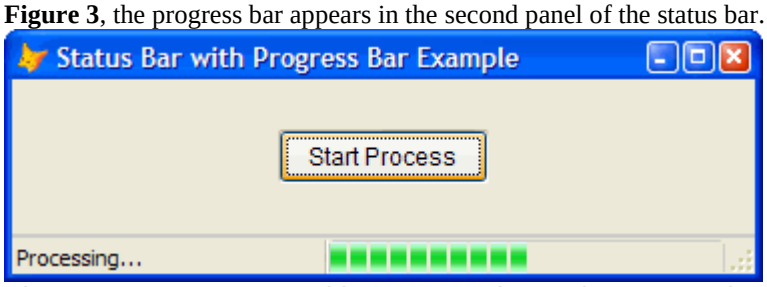

Figure 3. SFStatusBar combines a status bar and a progress bar in one control.

This form has the following code in Init to create a couple of panels in the status bar and put the progress bar in panel 2:

```
with This.oStatus
  .AddPanel('Ready', 300, .T., 0)
  .AddPanel('', 200, .F., 1)
  .RenderPanels()
  .ProgressBar.Panel = 2
endwith
```

The Click method of the button uses a dummy loop to demonstrate the progress bar.

```
* Have status panel 1 indicate what we're doing.

Thisform.oStatus.UpdatePanel(1, 'Processing...')

* Perform a loop and show the progress.

for lnI = 1 to 100
  Thisform.oStatus.ProgressBar.Value = lnI
  inkey(0.02, 'H')
next lnI

* Reset the progress bar back to 0 and update status
* panel 1.

Thisform.oStatus.ProgressBar.Value = 0
Thisform.oStatus.UpdatePanel(1, 'Ready')
```

Summary

This month, we looked at three of the tools Craig Boyd has generously donated to the VFP community: an encryption library and splitter and progress bar controls. Next month, we'll look at controls providing a VFP-based calendar, scrollbars, and task panels.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, and the MemberData Editor, Anchor Editor, New Property/Method Dialog, and CursorAdapter and DataEnvironment builders that come with VFP. He is co-author of the "What's New in Visual FoxPro" series and "The Hacker's Guide to Visual FoxPro 7.0," all from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over the world. He is a long-time Microsoft Most Valuable Professional (MVP), having first been honored with this award in 1996. Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com