November, 2002

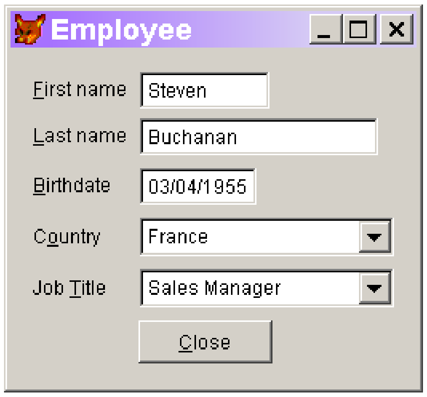
Let Users Control Fonts

Tamar E. Granor, Ph.D.

My eyesight isn't what it used to be. For example, reading the date on my wristwatch now requires a delicate maneuver to position it at just the right distance. Fortunately, most of the software I use every day provides easy ways to handle this problem. Some applications (like Word) let me zoom in and out, while others (like VFP and Windows) let me choose the fonts and sizes they use. In fact, it's reached the point where I'm pretty annoyed at apps that don't let me configure fonts and sizes to fit my needs.

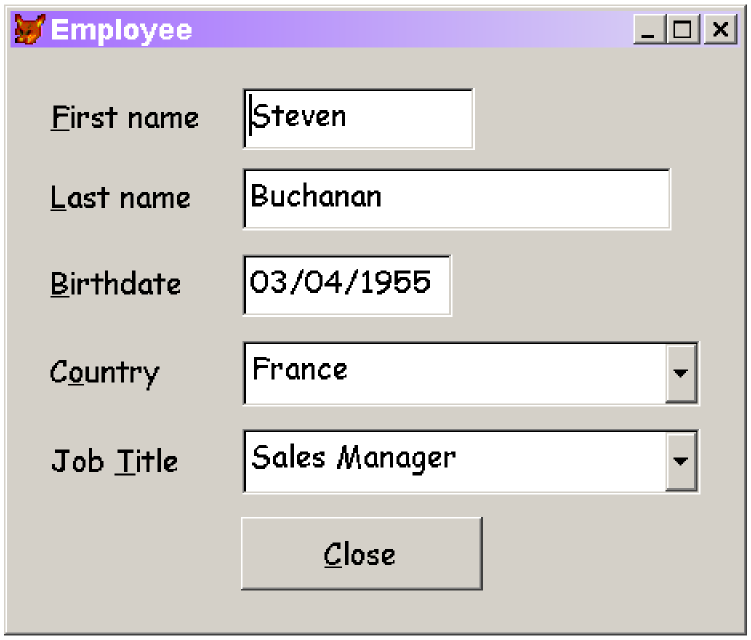
While I know I'm not a typical user by any stretch of the imagination, this is one area where I feel confident that lots of users share my feelings. No one wants to squint at a form to try to make out its contents, and no one font and size combination is going to be just right for all users. However, while Windows provides a way for users to control the colors used in applications, the font choices you can make in the Display Properties dialog apply only to Windows components (like title bars and menus) and not to the contents of windows.

So I set out to find a way to let my users configure fonts for forms in the applications I write. I ended up with three cooperating class hierarchies, one for each of the three major tasks involved in the process. The first class does the actual "refonting" of forms and their contents. The second handles resizing and repositioning of objects based on their new font characteristics. The final class in the group saves and restores font and position information, so that users only have to set it up once. As a strong believer in reuse, I was happy to adapt the latter two classes from existing classes written by other developers. Figure 1 shows an example form as it was originally created. Figure 2 shows the same form after changing the font to 14-point Comic Sans.



First name Steven
Last name Buchanan
Birthdate 03/04/1955
Country France
Job Title Sales Manager

Figure 1. As packaged–This example form was created using 9-point Arial.



First name Steven
Last name Buchanan
Birthdate 03/04/1955
Country France
Job Title Sales Manager

Figure 2. Bump it up–After changing the form's font to 14-point, Comic Sans, everything, including the form, is bigger.

Decomposing the problem

At first glance, the question of changing the font and size combination on a form sounded quite simple: Let the user choose a font and size and use the SetAll method to apply it to every object on the form. However, a little thought and experimentation turned up several problems:

- It's possible for different items on a form to use different font sizes in the first place. Those distinctions should be maintained. That is, if one control initially uses a 20-point font while another uses 10-point, there's probably a reason related to the relative importance of the two controls. Making both controls use the same font size could have a negative impact on the user interface. (It can also be the case that different controls use different fonts; the technique described here doesn't provide a way to maintain those differences, but could be adapted to do so.)

- When a font or font size is changed, the data may no longer fit into the allocated space. That's, of course, particularly an issue when the font size is increased, but it can also occur due to the choice of font. Even in the same size, some fonts are wider and others are narrower. Once we change the size of controls to accommodate the new font and size combination, controls might land on top of each other.

- Once a user changes the font for a form, she'll probably expect that form to use her choices every time she works with it (or, at least, until she changes it again). Users are likely to be pretty annoyed by an application that makes them set up their font choices every time they run a form.

Not coincidentally, the three problems map directly to the three class hierarchies I ended up with. The font handler class deals with the issue of maintaining the relative size of fonts when the font changes. The resizing class ensures that controls are the right size for their content, and that they don't overlap. Finally, the persistence classes maintain the user's settings.

Recursion and drilling down

All three classes use a similar strategy in performing their assigned tasks. They start at the form-level and drill down to every contained control. Each of the class hierarchies has at least one recursive method. (A recursive method is one that calls itself).

VFP supports recursion happily, as long as you don't exceed the program stack limit of 128 calls. Allowing a few levels for setting things up, a VFP form that has controls nested more than 120 levels is extremely unlikely. While the total number of objects on a form may reach that range, to cause a problem with the recursive methods, they'd have to be nested that deeply.

Drilling down into VFP's containership hierarchy is quite simple. In VFP 7, every container object has an Objects collection with a member for each contained object. That means you can write code like this:

```
* Assume oCurrentControl is the control being processed
* Assume also that we're in a method called SomeMethod
FOR EACH oControl in oCurrentControl.Objects
   This.SomeMethod( oControl )
ENDFOR
```

In earlier versions of VFP, you have to drill down through various different collections. For example, pageframes have a Pages collection, while OptionGroups and CommandGroups have Buttons collections. Containers (like forms and pages) that can hold objects of different types have a Controls collection. So, code meant to run in versions prior to VFP 7 has to use a case statement along these lines to drill down:

```
* Same assumptions as above
DO CASE
CASE UPPER(oCurrentControl.BaseClass) = "PAGEFRAME"
   FOR EACH oPage IN oCurrentControl.Pages
      This.SomeMethod( oPage )
   ENDFOR
CASE INLIST(UPPER(oCurrentControl.BaseClass), ;
            "OPTIONGROUP", "COMMANDGROUP")
   FOR EACH oButton IN oCurrentControl.Buttons
      This.SomeMethod( oButton )
   ENDFOR
* Other cases to handle other types of controls
OTHERWISE
   FOR EACH oControl IN oCurrentControl.Controls
      This.SomeMethod( oControl )
   ENDFOR
ENDCASE
```

The actual processing component of a particular method can occur before or after such a loop. In fact, the font handler code processes containers first, then drills down into their members, while the resizing code drills down first and resizes the container after its members have been processed.

Changing Fonts

The cusFontHandler class allows a user to choose a font and size combination and applies it proportionally to all controls on the form. As written, the class can actually change the font for any container (not just a form) and its contents. It has a number of custom methods and properties, shown in Tables 1 and 2.

Table 1. Font Handler properties–The font handling class has these custom properties.

Property	Purpose
cDefaultFont	The default font to display when prompting the user for the new font, if the target container doesn't have font properties.
cFontName	The name of the font chosen by the user.
cResize	The name of a class to use for resizing and repositioning controls.
cResizeLib	The name of the class library containing the resizer class.
lDrillDown	Indicates whether the new font and size should be applied to contained objects. .T., by default.
lNoResize	Indicates whether the resizing step should be omitted. .F., by default.
nDefaultSize	The default font size to display when prompting the user for the new font, if the target container doesn't have font properties.
nFontSize	The size of the font chosen by the user.
nOriginalSize	The original font size of the container (form), used in proportional sizing.
oResize	An object reference to the resizing object.
oTarget	An object reference to the object (form) being refonted.

Table 2. Font Handler methods–The font handling class has a number of custom methods. Only the ChangeFont method is public; the rest are protected.

Method	Purpose
ApplyFont	Applies the new font and size to an object and then, recursively, to its contents.

Method	**Purpose**
ApplyFontToGrid	Applies the new font and size to a grid and its contents. Grids require special handling.
ChangeFont	The key method that lets the user choose a new font and size and apply it.
ComputeNewFontSize	Computes the new font size of a control, given the old font size.
GetFont	Sets things up for the user to choose a new font and size.
PromptForFont	Prompts the user to choose a new font and size. Called by GetFont.

In addition to the methods listed in Table 2, the Init and Destroy methods of the class contain code to instantiate and destroy, respectively, the specified resizer class.

The ChangeFont method is the only public method among the custom methods. To change the font for a container and its contents, call ChangeFont, passing an object reference to the target object (the object whose font is to be changed). ChangeFont also accepts two optional parameters that can override the settings of lDrillDown and lNoResize.

ChangeFont doesn't actually do the work of changing fonts; it calls on other methods of the class. Here's the main body of ChangeFont:

```
IF lReturn
   IF This.GetFont()
      IF NOT This.lNoResize AND ;
         NOT PEMSTATUS(This.oTarget, "lHasResizerData", 5)
         * If we're resizing and haven't saved form info,
         * save it now
         This.cusResize.SaveFormDimensions( This.oTarget )
      ENDIF
      This.ApplyFont( This.oTarget )
      * Now resize if requested and available
      IF NOT This.lNoResize AND ;
         PEMSTATUS(This,"cusResize", 5)
         * If we're resizing, do it.
         This.cusResize.AdjustControls( This.oTarget )
      ENDIF
   ENDIF
```

```
ENDIF
```

The code here can be divided into four parts: choosing a font, preparing for resizing, applying the new font, and resizing. We'll look at choosing a font and applying it here, and examine the two resizing tasks later on.

Choosing a Font

The process of letting the user choose a new font is divided into two parts. The GetFont method stores values needed for font calculations, and then it calls PromptForFont to let the user actually select the new font. The code for GetFont is quite simple:

```
* Let the user choose a font.
LOCAL lReturn

* First, set up defaults
IF PEMSTATUS(This.oTarget,"FontName",5)
   This.cFontName = This.oTarget.FontName
   This.nFontSize = This.oTarget.FontSize

   This.nOriginalSize = This.nFontSize
ELSE
   This.cFontName = This.cDefaultFont
   This.nFontSize = This.nDefaultSize
   This.nOriginalSize = This.nFontSize
ENDIF

lReturn = This.PromptForFont()

RETURN lReturn
```

In the class provided on this month's Professional Resource CD and on Advisor.COM, PromptForFont calls VFP's built-in GETFONT() function and parses the results, storing them in properties of the font handler. However, you may prefer a different interface for choosing a font (especially because the dialog displayed by GETFONT() allows the user to choose font characteristics such as bold and italic that are ignored by the class). In fact, that's why this part of the process was separated into another method. A subclass can simply override PromptForFont. Here's the code that uses GETFONT().

```
* Prompt the user for font input
* and store the results in the right properties

* In this version, the GETFONT() dialog is used
* but the bold and italic settings are ignored.

LOCAL cNewFont, lReturn
```

```
cNewFont = GETFONT(This.cFontName, This.nFontSize, "" )

IF NOT EMPTY(cNewFont)
   ALINES(aFontInfo, cNewFont, ",")
   This.cFontName = aFontInfo[1]
   This.nFontSize = VAL(aFontInfo[2])
   lReturn = .T.
ELSE
   lReturn = .F.
ENDIF

RETURN lReturn
```

Applying the font

The heart of the font handler class is the methods ApplyFont and ApplyFontToGrid. That's where the actual work of changing fonts and sizes occurs.

ApplyFont uses a top-down approach, changing the font of an object, then calling itself recursively for all the objects contained within the current object. Here's the code:

```
* Apply the newly chosen font to the object
* and, if appropriate, to its contained objects.
LPARAMETERS oApplyTo

LOCAL nRatio

IF PEMSTATUS(oApplyTo, "FontName",5)
   WITH oApplyTo
      * Compute new font size
      nNewSize = This.ComputeNewFontSize(.FontSize)
      IF nNewSize < 8
         * Handle small fonts
         .FontSize = nNewSize
      ENDIF
      .FontName = This.cFontName
      .FontSize = nNewSize
   ENDWITH
ENDIF

IF This.lDrillDown
   * Apply the new font to all contained objects
   * keeping the fonts proportional

   IF PEMSTATUS(oApplyTo, "Objects", 5)
      * it's a container, so away we go
      FOR EACH oObject IN oApplyTo.Objects
         IF UPPER(oObject.BaseClass) = "GRID"
            This.ApplyFontToGrid( oObject )
         ELSE
```

```
            This.ApplyFont( oObject )
         ENDIF
      ENDFOR
   ENDIF
ENDIF

RETURN
```

The new font is simply applied to the object. The new font size is computed by the ComputeNewFontSize method, based on the current font size, the size chosen by the user, and the form's font size at the start of the refonting process (stored in the nOriginalSize property by GetFont). Here's the code:

```
* Compute the new font size for a given font size
LPARAMETERS nOldSize

LOCAL nRatio, nNewSize

ASSERT VARTYPE(nOldSize) = "N" MESSAGE ;
   "ComputeNewFontSize: Must pass numeric parameter"

nRatio = nOldSize/This.nOriginalSize
nNewSize = ROUND(This.nFontSize * nRatio,0)

RETURN nNewSize
```

Grids turn out to need special handling, however. When you change the font and size of a grid or column, the changes are automatically passed along to the members of the grid or column. So, for grids only, a two-step approach is needed in order to ensure that changes get applied correctly. The ApplyFontToGrid method uses another class (cusTraverseGrid) to build an array containing an entry for each object contained in the grid (drilling down to the very bottom level). The method then saves the current font size for each object into another array. At that point, it applies the new font and size to the grid, then uses the array of font sizes to compute the new sizes for all objects contained in the array. In this way, the code doesn't depend on the objects inside the grid to have their old font information at the time their new size is computed. Here's the code:

```
* Change font in grid. Special handling is needed
* because any change to a grid or column's FontName
* and FontSize properties is automatically passed down to its members.
LPARAMETERS oGrid AS Grid

LOCAL aGridData[1], oTraverser, oColumn AS Column,
LOCAL nEntries, nCount, aFontData[1]

oTraverser = NEWOBJECT("cusTraverseGrid","Accessibility")
```

```
aGridData = oTraverser.BuildGridArray(oGrid)
RELEASE oTraverser

nEntries = ALEN(aGridData)
DIMENSION aFontData[ nEntries, 2]

* Now get original font info
FOR nCount = 1 to nEntries
   aFontData[ nCount, 1 ] = aGridData[ nCount ]
   IF PEMSTATUS(aGridData[ nCount ], "FontSize", 5)
      aFontData[ nCount, 2 ] = ;
         aGridData[ nCount ].FontSize
   ELSE
      aFontData[ nCount, 2 ] = -1
   ENDIF
ENDFOR

* Now apply font
* First, apply to grid
oGrid.FontName = This.cFontName
oGrid.FontSize = This.ComputeNewFontSize(oGrid.FontSize)

* Now apply to grid members from the outside in
* using stored font information

FOR nCount = 1 to nEntries
   IF aFontData[ nCount, 2] <> -1
      aFontData[ nCount, 1].FontName = This.cFontName
      aFontData[ nCount, 1].FontSize = ;
         This.ComputeNewFontSize( aFontData[ nCount, 2] )
   ENDIF
ENDFOR
```

Be aware that most ActiveX controls aren't affected by this class, since, in general, they don't have FontName or FontSize properties. However, many ActiveX controls use a contained Font object that has Name and Size properties. You can add code to ApplyFont to look for this object and manipulate its properties.

Resizing and Repositioning

With the font-handling piece in place, I needed a way to adjust the form once the controls' fonts had been changed. I knew I'd seen a number of classes over the years designed to handle resizing and repositioning, so I didn't want to start from scratch. As I looked around, I found that those classes were designed to respond when a user resized a form, moving and sizing controls appropriately, possibly changing font sizes.

My task was the converse, take the form with changed fonts and resize and reposition the controls. However, I found a class written by Marcia

Akins (published in her book "1001 Things You Wanted to Know about Visual FoxPro") that offered a good strategy for the problem. With her permission, I created the cusResizer class based on (but not subclassed from) her class of the same name.

The cusResizer class has two custom properties and a number of custom methods, shown in Tables 3 and 4.

Table 3. Resizer properties–The cusResizer class has only two custom properties. However, it adds properties to the objects it's resizing.

Property	**Purpose**
lResizeForm	Indicates whether the form itself is to be resized. If not, scroll bars are enabled.
oTarget	An object reference to the form to be resized.

Table 4. Resizer methods–Only the AdjustControls and SaveFormDimensions methods are public. The rest are protected and called by those two.

Method	**Purpose**
AdjustControls	The public method that kicks off the process of resizing and repositioning controls.
GetControlWithFont	Starting with an object passed as a parameter, climbs the containership hierarchy until it finds an object that has font properties.
GetFontHeight	Computes the average height of a character in the font used by a control.
GetFontWidth	Computes the average width of a character in the font used by a control.
GetHeightRatio	Computes the ratio between the height of a control's original font and the height of the new font.
GetWidthRatio	Computes the ratio between the width of a control's original font and the width of the new font.

Method	**Purpose**
ResizeControls	Performs the actual resizing and repositioning of controls. Called recursively to drill down into the form.
SaveFormDimensions	Stores the original dimensions of a form and its controls
SaveOriginalDimensions	Stores the original dimensions of a control. Called recursively to handle contained controls.

While changing the font lent itself to a top-down approach, resizing and repositioning must be performed from the bottom up. That's because the size of a container is dependent on the size of its contents. For some controls, that's a VFP requirement. For example, the pages in a pageframe must all be the same size, and it's set at the pageframe level. To set the pageframe's Height and Width, you need to know the maximum size of an individual page.

Preparing for resizing

The goal in resizing and repositioning the controls is to maintain the original proportions of the form. That means that the new size and position of a control needs to be calculated with respect to the original size and position of that control. Since a form and its controls may be resized and repositioned many times, we need a way to track the original dimensions of each object.

What made Marcia's resizer class so appealing, in fact, is its clever solution to this problem. The class dynamically adds properties to each object on the form (using the AddProperty method) to hold its own original size and position. This avoids having to manage some kind of storage (like an array or table) for the data.

The SaveFormDimensions method kicks this process off by storing form-level information. It calls the SaveOriginalDimensions method for each control on the form. The methods store the Top, Left, Height, Width and FontSize for each control that has those properties. For container controls, they also store the ratio between the rightmost and bottommost position of a container control and the width and height of the container. The ratios are used when computing the new size of the container after resizing and repositioning its contents.

Here's the code for SaveFormDimensions:

```
* Save form and control data.
LPARAMETERS toForm

ASSERT VARTYPE(toForm) = "O" OR ;
       UPPER(This.Parent.BaseClass) = "FORM" ;
   MESSAGE "SaveFormDimensions: This object must be " + ;
       "placed on a form or receive a form as parameter"


IF VARTYPE(toForm) <> "O" AND ;
   UPPER(This.Parent.BaseClass) <> "FORM"
   ERROR 11
   RETURN .F.
ENDIF

LOCAL loControl

IF VARTYPE(toForm) = "O"
   This.oTarget = toForm
ELSE
   This.oTarget = ThisForm
ENDIF

WITH This.oTarget
   * Save form dimensions and font info at instantiation
   .AddProperty( 'lHasResizerData', .T. )
   .AddProperty( 'nOriginalHeight', .Height )
   .AddProperty( 'nOriginalWidth', .Width )
   .AddProperty( 'nOriginalFontSize', .FontSize )
   nFontWidth = This.GetFontWidth( This.oTarget )
   nFontHeight = This.GetFontHeight( This.oTarget )
   .AddProperty( 'nOriginalFontWidth', nFontWidth)
   .AddProperty( 'nOriginalFontHeight', nFontHeight)

   * Set a minimum Width and Height to avoid errors later
   .MinWidth = .Width / 2
   .MinHeight = .Height / 2

   * Now save the relevant visual properties
   * (height, width, columnwidths, etc)
   * of all the controls on the form
   nRightMost = 0
   nBottomMost = 0

   FOR EACH loControl IN .Controls
      This.SaveOriginalDimensions( loControl )
      nRightMost = MAX(nRightMost, loControl.Left + loControl.Width)
      nBottomMost = MAX(nBottomMost, loControl.Top + loControl.Height)
   ENDFOR
   * Save relationship of far edges to form
   nWidthRatio = .Width/nRightMost
   nHeightRatio = .Height/nBottomMost
   .AddProperty( "nOriginalWidthRatio", nWidthRatio)
   .AddProperty( "nOriginalHeightRatio", nHeightRatio)

ENDWITH
```

SaveOriginalDimensions performs a similar task for the individual controls. Because different controls have different needs, it uses a case statement based on the control's BaseClass. Here's a portion of the code, showing the information saved for all controls and the cases for pageframes and pages.

```
LPARAMETERS toControl
LOCAL loPage, loControl, loColumn, lnCol
LOCAL cStyle, nFontWidth, nFontHeight
LOCAL nRightMost, nBottomMost

* If the object does not have an AddProperty method, we
* can't add the properties to save the original dimension.
* So bail out
IF ! PEMSTATUS( toControl, 'AddProperty', 5 )
   RETURN
ENDIF

* Add the properties to hold the object's
* original dimensions
IF PEMSTATUS( toControl, 'Width', 5 )
   toControl.AddProperty( 'nOriginalWidth', ;
                          toControl.Width )
ENDIF
IF PEMSTATUS( toControl, 'Height', 5 )
   toControl.AddProperty( 'nOriginalHeight', ;
                          toControl.Height )
ENDIF
IF PEMSTATUS( toControl, 'Top', 5 )
   toControl.AddProperty( 'nOriginalTop', ;
                          toControl.Top )
ENDIF
IF PEMSTATUS( toControl, 'Left', 5 )
   toControl.AddProperty( 'nOriginalLeft', ;
                          toControl.Left )
ENDIF
IF PEMSTATUS( toControl, 'Fontsize', 5 )
   toControl.AddProperty( 'nOriginalFontSize', ;
                          toControl.FontSize )
   nFontWidth = This.GetFontWidth( toControl )
   nFontHeight = This.GetFontHeight( toControl )
   toControl.AddProperty( 'nOriginalFontWidth', ;
                          nFontWidth)
   toControl.AddProperty( 'nOriginalFontHeight', ;
                          nFontHeight)
```

```
ENDIF

* Now see if we have to drill down. Also, take care
* of special cases like grids where we have to save
* RowHeight, HeaderHeight, and combos where we need
* to save ColumnWidths
DO CASE
CASE UPPER( toControl.BaseClass ) = 'PAGEFRAME'
   nRightMost = 0
   nBottomMost = 0

   FOR EACH loPage IN toControl.Pages
      This.SaveOriginalDimensions( loPage )
      FOR each loControl IN loPage.Controls
         nRightMost = MAX(nRightMost, ;
            loControl.Left + loControl.Width)
         nBottomMost = MAX(nBottomMost, ;
            loControl.Top + loControl.Height)
      ENDFOR
   ENDFOR

   * Save relationship of far edges to controls
   nWidthRatio = toControl.Width/nRightMost
   nHeightRatio = toControl.Height/nBottomMost
   toControl.AddProperty( "nOriginalWidthRatio", ;
                          nWidthRatio)
   toControl.AddProperty( "nOriginalHeightRatio", ;
                          nHeightRatio)


CASE INLIST( UPPER( toControl.BaseClass ), 'PAGE')
   FOR EACH loControl IN toControl.Controls
      This.SaveOriginalDimensions( loControl )
   ENDFOR

  * Other cases for other containers
```

Resizing

The AdjustControls method starts the process of resizing and repositioning. It loops through all the controls on the form, applying the ResizeControls method to each. It also tracks the rightmost and bottommost position of the controls once they're sized and moved.

Finally, if the form itself is to be resized (the lResizeForm property is .T.) , the new height and width are calculated based on the rightmost and bottommost positions found and the original height and width ratios. If resizing of the form is turned off, scrollbars are added to the form as needed. (However, due to an optimization in the VFP engine, scrollbars actually appear only if the form's ScrollBars property was not 0 at the time the form was instantiated.)

Here's the code for AdjustControls:

```
* Adjust sizes for specified form
LPARAMETERS toForm

ASSERT VARTYPE(toForm) = "O" OR ;
       UPPER(This.Parent.BaseClass) = "FORM" ;
   MESSAGE "AdjustControls: This object must be " + ;
      "placed on a form or receive a form as parameter"

IF VARTYPE(toForm) <> "O" AND ;
   UPPER(This.Parent.BaseClass) <> "FORM"
   ERROR 11
   RETURN .f.
ENDIF

IF VARTYPE(toForm) = "O"
   This.oTarget = toForm
ELSE
   This.oTarget = ThisForm
ENDIF

LOCAL loControl, nRightMost, nBottomMost
This.oTarget.LockScreen = .T.

nRightMost = 0
nBottomMost = 0
FOR EACH loControl IN This.oTarget.Controls
   This.ResizeControls( loControl )
   nRightMost = MAX(nRightMost, ;
      loControl.Left + loControl.Width)
   nBottomMost = MAX(nBottomMost, ;
      loControl.Top + loControl.Height)
ENDFOR

* Now deal with the form itself
IF This.lResizeForm
   WITH This.oTarget
      .Width = nRightMost * .nOriginalWidthRatio
      .Height = nBottomMost * .nOriginalHeightRatio
      .Scrollbars = 0
   ENDWITH
ELSE
   nScrollBars = 0
   IF nRightMost > This.oTarget.Width
      nScrollBars = nScrollBars + 2
   ENDIF
   IF nBottomMost > This.oTarget.Height
      nScrollBars = nScrollBars + 1
   ENDIF
   This.oTarget.ScrollBars = nScrollBars
ENDIF

This.oTarget.LockScreen = .F.
```

Most of the work of resizing and repositioning is done by the ResizeControls method. Like SaveOriginalDimensions, it has separate cases for the different types of containers. The code for some containers, like pageframes and the Container baseclass, looks much like the form-level code in AdjustControls. Other cases require a different approach. Here's the case for a grid:

```
CASE UPPER( toControl.BaseClass ) = 'GRID'
   IF PEMSTATUS( toControl, 'Width', 5 )
      toControl.Width = toControl.nOriginalWidth * ;
         This.GetWidthRatio( toControl)
   ENDIF
   IF PEMSTATUS( toControl, 'Height', 5 )
      toControl.Height = toControl.nOriginalHeight * ;
         This.GetHeightRatio( toControl)
   ENDIF

   WITH toControl
      .RowHeight    = .nOriginalRowHeight * ;
         This.GetHeightRatio( toControl)
      .HeaderHeight = .nOriginalHeaderHeight * ;
         This.GetHeightRatio( toControl)
      FOR lnCol = 1 TO .ColumnCount
         .Columns[ lnCol ].Width = ;
            .nOriginalColumnWidths[ lnCol ] * ;
            This.GetWidthRatio( toControl)
      ENDFOR
   ENDWITH
```

And here's the final case, which handles non-container controls:

```
OTHERWISE
   * Handle "scalar" controls
   IF PEMSTATUS( toControl, 'Width', 5 )
      toControl.Width = toControl.nOriginalWidth * ;
         This.GetWidthRatio( toControl)
   ENDIF
   IF PEMSTATUS( toControl, 'Height', 5 )
      toControl.Height = toControl.nOriginalHeight * ;
         This.GetHeightRatio( toControl)
   ENDIF
```

The GetHeightRatio and GetWidthRatio methods compute the ratio of the new font's height or width to the same characteristic of the original font. These methods call on the GetFontHeight and GetFontWidth methods, respectively, to compute the height and width of the specified font in the specified size with the control's current characteristics (such as bold, italic, etc.).

Here's the code for GetHeightRatio. GetWidthRatio is analogous.

```
* Compute the Height ratio of the new size to the old
* for a control
LPARAMETERS oControl

ASSERT VARTYPE(oControl) = "O" ;
   MESSAGE "GetHeightRatio: Must pass object"
IF VARTYPE(oControl) <> "O"
   ERROR 11
   RETURN -1
ENDIF

LOCAL nFontHeight, nRatio, oUseControl

* This control may not have font properties, so move up
* until you find one that does. Since we're on a form,
* there'll always be one.
oUseControl = This.GetControlWithFont(oControl)

WITH oUseControl
   nFontHeight = This.getfontheight(oUseControl)
   nRatio = nFontHeight/.nOriginalFontHeight
ENDWITH

RETURN nRatio
```

Here's GetFontHeight. Again, GetFontWidth is analogous.

```
* Compute the average Height of a character
* for a specified control
LPARAMETERS oControl

ASSERT VARTYPE(oControl) = "O" ;
   MESSAGE "GetFontHeight: Must pass object"
IF VARTYPE(oControl) <> "O"
   ERROR 11
   RETURN -1
ENDIF

WITH oControl
   cStyle = IIF(.FontBold,"B","")
   cStyle = IIF(.FontItalic,cStyle + "I",cStyle)
   cStyle = IIF(.FontOutline,cStyle + "O",cStyle)
   cStyle = IIF(.FontShadow,cStyle + "S",cStyle)
   cStyle = IIF(.FontStrikethru,cStyle + "-",cStyle)
   cStyle = IIF(.FontUnderline,cStyle + "U",cStyle)
   nFontHeight = FONTMETRIC(1, .FontName, ;
                            .FontSize, cStyle )

ENDWITH

RETURN nFontHeight
```

Putting the font handler to work

The next piece to letting users change fonts on the fly is some mechanism for starting the process. One common way to give users this ability is with a context (right-click) menu. The PRD and downloads include a context menu that contains just a Fonts item. That item calls a program, SetFont, like this:

```
DO SetFont WITH _SCREEN.ActiveForm
```

The SetFont program makes sure there's a cusFontHandler object available and calls its ChangeFont method passing a reference to the active form. SetFont is written so that it can work in the context of an application object (referenced by oApp) or on its own:

```
* Call on the Font Handler to change the font for the
* passed object. If there's an application object,
* look for the Font Handler there. Otherwise, use
* a public variable.
LPARAMETERS oTarget

* Make sure we have a target
IF VARTYPE(oTarget) <> "O"
   ERROR 11
   RETURN .F.
ENDIF

LOCAL oFonts

DO CASE
CASE TYPE("oApp") = "O" AND ;
     PEMSTATUS(oApp, "oFontHandler",5)
   IF VARTYPE(oApp.oFontHandler) <> "O"
      * instantiate it
      oApp.oFontHandler = ;
               NEWOBJECT("cusFontHandler","Accessibility")
   ENDIF
   oFonts = oApp.oFontHandler

CASE VARTYPE(oFontHandler) <> "O"
   RELEASE oFontHandler
   PUBLIC oFontHandler

   oFontHandler = NEWOBJECT("cusFontHandler",;
                              "Accessibility")
   oFonts = oFontHandler

OTHERWISE
   * it already exists in the public variable
   oFonts = oFontHandler
ENDCASE
```

```
oFonts.ChangeFont( oTarget, .T. )

RETURN
```

Making it stick

Once the user has adjusted the font for a form, he probably wants to continue using that font/size combination. So the final piece of the puzzle is the ability to store the font settings and restore them when the form is next run.

This part of the task uses classes based on a set written by Doug Hennig. Doug's classes combined the process of deciding what to store with the actual storage mechanism. I refactored his classes into two class hierarchies. sfPersistent (which has the same name as Doug's original root class, but is somewhat modified) is an abstract class that does everything except actually store the data. It has an abstract DefineItems method that lets subclasses specify which items to store.

PersistentStorage is an abstract class for the storage of data. It has SaveOne and RestoreOne methods to hold the code that actually stores and retrieves data. sfPersistent has properties (cStorageClass, cStorageClassLib) to tell it which PersistentStorage subclass to use – it instantiates the appropriate class in its Init method.

To actually store the font data, I use a subclass of sfPersistent called cusPersistFonts. This class has a custom property, cFormID, that you must fill in for each form that uses the control. The property identifies the form, so that you can store font data for multiple forms in the same table. I tend to use the name of the form.

cusPersistFonts has code in the DefineItems method that tells it to store the Left, Top, Height, Width, FontName and FontSize properties for the form itself and every control on it. Here's the code:

```
* Add the font and size data to the list to be preserved
LPARAMETERS toObject

LOCAL lcObjectId, lcObjectPath, lcRelativeObjectPath
LOCAL loObject, loSubObject

* We need a form id
IF EMPTY(This.cFormID)
   RETURN .F.
ENDIF

* If no object passed, work with the form.
IF VARTYPE(toObject) = "O"
```

```
   loObject = toObject
ELSE
   loObject = ThisForm
ENDIF

* Skip this object
IF loObject = THIS
   RETURN
ENDIF

lcObjectPath = SYS(1272, loObject)
IF "." $ lcObjectPath
   lcRelativeObjectPath = "ThisForm." + ;
     SUBSTR(lcObjectPath,AT(".",lcObjectPath) + 1)
   lcObjectId = This.cformid + "." + ;
      SUBSTR(lcObjectPath,AT(".",lcObjectPath) + 1)
ELSE
   * This is the form
   lcRelativeObjectPath = "ThisForm"
   lcObjectId = This.cFormID
ENDIF

IF PEMSTATUS(loObject, "Left",5)
   This.AddItem(lcObjectId + ".Left", lcRelativeObjectPath + ".Left")
ENDIF

IF PEMSTATUS(loObject, "Top",5)
   This.AddItem(lcObjectId + ".Top", lcRelativeObjectPath + ".Top")
ENDIF

IF PEMSTATUS(loObject, "Height",5)
   This.AddItem(lcObjectId + ".Height", lcRelativeObjectPath + ;
      ".Height")
ENDIF

IF PEMSTATUS(loObject, "Width",5)
   This.AddItem(lcObjectId + ".Width", lcRelativeObjectPath + ".Width")
ENDIF

IF PEMSTATUS(loObject, "FontName",5)
   This.AddItem(lcObjectId + ".FontName", lcRelativeObjectPath + ;
       ".FontName")
ENDIF

IF PEMSTATUS(loObject, "FontSize",5)
   This.AddItem(lcObjectId + ".FontSize", lcRelativeObjectPath + ;
      ".FontSize")
ENDIF

* Now handle contained objects
IF PEMSTATUS(loObject, "Objects", 5)
   FOR EACH loSubObject IN loObject.Objects
      This.DefineItems(loSubObject)
   ENDFOR
ENDIF
```

```
RETURN
```

I chose to store the font data in a table, so cusPersistFonts specifies a subclass of PersistentStorage called PersistentTable. (There's also a subclass for storing data in the Registry.) PersistentTable has a custom property to specify the name of the table in which to store the data, as well as code in its Init method to make sure the table exists and is open. The SaveOne and RestoreOne methods are fairly simple, finding or creating the appropriate record and storing or retrieving the data.

The original form settings can be restored by deleting the whole table or just the records that relate to a particular form. You might want to add a "Restore Font Settings" item to the context menu or to the dialog for choosing fonts (if you're using something other than the GETFONT() function) and have it perform this action.

This set of classes doesn't actually interact with the font handling and resizing classes. To set up a form so that its font information is saved, just drop an instance of cusPersistFonts onto the form. Be sure it’s the last control to be instantiated, however; since DefineItems is called from the Init method, you need to be sure all the other objects have been instantiated and can be found when DefineItem runs. To make it the last control, click on it in the Form Designer and choose Format-Bring to Front.

Give Them What They Want

Even when you put control over fonts and sizes into your user's hands, you still need to spend time properly designing forms so they're visually attractive and have an appropriate flow of control. However, you don't have to worry about finding the ideal trade-off of space used vs. readability. Each user can choose that for himself.

Complete code for all of the classes described in this article is in included on this month's Professional Resource CD and can be downloaded from Advisor.COM. There's also a sample form demonstrating changing fonts.