November, 2001

Advisor Answers

Detecting Windows color changes

Visual FoxPro 7.0/6.0/5.0/3.0

Q: I'm using the GetSysColor API function to retrieve the user's colors for some special highlighting in my application. But, if the user changes his color scheme while my forms are running, those colors don't change. Is there a way to make my forms respond to system color changes?

–Name withheld by request

A: As you know, it's a good idea to design forms to use the colors specified by the user (or the system administrator, in some cases) in the Control Panel's Display Properties dialog. While many Windows users simply leave the default color settings (and may, in fact, be unaware that they can change them), other users, such as those with color blindness, choose colors that make Windows usable for them. Ignore those settings at your own risk, as hard-coding colors in your application may make it inaccessible to some users.

For ordinary circumstances, it's easy to honor the user's color choices in VFP. Just make sure the ColorSource property of your forms is set to either 4 or 5 (Windows dialog colors and Windows document colors, respectively) and that ColorSource for controls is set to 4 (Windows colors). Don't set any of the other color-related properties.

But what happens when you need to do something different than the default? For example, you might want to make a particular control stand out in some way. In that case, as you note, the key is to use the GetSysColor API function to extract appropriate colors from the user's color scheme and apply them. I wrote about GetSysColor in the July, 1999 ADVISOR Answers column, so I won't repeat the details here.

This approach leaves one small problem. When VFP forms use the Windows colors, they change colors whenever the user changes the color scheme. However, when you hard-code a color, even using GetSysColor, that element isn't automatically changed.

To make matters worse, the only form event that fires when the system colors are changed is Paint. However, because Paint fires so often, we want to minimize the amount of code there, and execute only what we need to.

Nonetheless, we can put code in Paint to update the colors. To minimize the code, we'll check whether any action is necessary before we take it.

However, there's another approach that's more elegant and has our code execute only when necessary. The SysInfo ActiveX control has events that fire for all kinds of system actions. The one we're interested in is SysColorsChanged. If we add a SysInfo control to the form, we can put code in that event to respond to color changes.

In the example form shown in Figure 1, we want a special background around the three buttons. One way to get it is to use the Windows highlight color to fill the shape. The highlight color is the color used for selected items (as in menus). To make the shape stand out even more, we'll set the border to use the highlight text color (used for selected menu items, among other things).

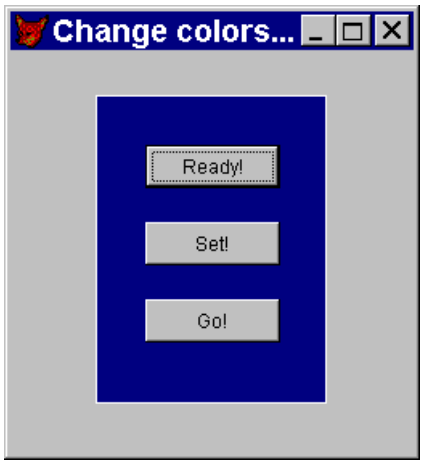

Figure 1. Setting colors based on color scheme–The shape behind the buttons draws its colors from the user's current color scheme, and changes them when the color scheme changes.

To implement this, add two new properties to the form, nSpecialFillColor and nSpecialBorderColor. Set them both to 0 initially.

In the form's Load event, we need to make the GetSysColor function available. Once we declare the function, it's available until we explicitly clear it with either CLEAR DLLS, CLEAR ALL, or, in VFP 7, CLEAR DLL GetSysColor.

```
DECLARE INTEGER GetSysColor IN Win32API INTEGER nIndex
```

In the SysColorsChanged event, we check to see whether the color properties are up-to-date. If not, update them and update the display:

```
* Check whether relevant system colors have changed
LOCAL nHighlightColor, nFillColor
nHighlightColor = GetSysColor(13)
nFillColor = GetSysColor(14)
```

```
IF nHighlightColor <> This.nSpecialFillColor or;
   nFillColor <> This.nSpecialBorderColor
   * Colors have changed. Save the new ones.
   This.nSpecialBorderColor = nFillColor
   This.nSpecialFillcolor = nHighlightColor

   * Fix the colors
   This.SetCustomColors()
ENDIF
```

Next, add a custom method, SetCustomColors, to set the control colors to use the colors stored in the properties. While this code could go into the SysColorsChanged event, isolating it into a custom event makes it easier to find if we want to add other controls to the list changed or change what we're doing.

```
This.shpBackground.FillColor = This.nSpecialFillColor
This.shpBackground.BorderColor = This.nSpecialBorderColor
```

Finally, add code to Init to set the special colors up initially:

```
* Set special colors
This.nSpecialBorderColor = GetSysColor(14)
This.nSpecialFillcolor = GetSysColor(13)

* Fix the colors
This.SetCustomColors()
```

Now, if the user changes the system colors, the form, including the shape, follows along. Figure 2 shows the example form as it appears if the user changes to the Spruce color scheme.

Figure 2. Same form, new colors – When the Spruce color scheme is chosen, the form follows suit, including the shape.

This month's PRD contains two versions of the example form. SetColorsX.SCX uses the SysInfo control. SetColors.SCX uses the Paint method. This version

doesn't need code in the Init method because Paint fires as the form starts up and ensures that the colors are right initially. Because code in the Paint method can slow down execution, use this version of the technique sparingly.

–Tamar