Handling Multiple Monitors

Doug Hennig

This is the first of several articles on components of Doug's in-house library. This issue focuses on handling multiple monitors when persisting the size and location of forms.

Like most developers, I have more than one monitor. My system is a laptop so I use the laptop display as the primary monitor and a 24" monitor at the right as the second one. I typically have browser and explorer windows open on the second monitor and keep the primary monitor for those things I live in all day long (VFP and Outlook, mostly). I'm more productive because I'm not digging through stacks of windows and constantly moving or resizing one window or another.

However, one of the things I discovered fairly soon after adding a second monitor is that some of my applications didn't respect it. For example, I have a class called SFPersistentForm (discussed in the January 2000 issue of FoxTalk, the predecessor to FoxRockX) that I drop on most of my forms. It saves the form size and position when the form is closed and restores it when the form is reopened, giving the user the experience they expect when working with that form. However, I discovered that if I opened a form and moved it to the second monitor then closed it, when I reopened it, the form displayed on the primary monitor instead (this was a form with Desktop set to .T. so it can exist outside the application's window).

I quickly found out why: the persistence code was trying to prevent the situation where the form may open outside the screen boundaries, making it invisible. The following code handled that:

```
Thisform.Width  = min(max(Thisform.Width, ;
    0, Thisform.MinWidth), _screen.Width)
Thisform.Height = min(max(Thisform.Height, ;
    0, Thisform.MinHeight), _screen.Height)
Thisform.Left   = min(max(Thisform.Left, ;
    0), _screen.Width  - 50)
Thisform.Top    = min(max(Thisform.Top, ;
    0), _screen.Height - 50)
```

("- 50" is used to ensure the form didn't start at exactly the right or bottom boundaries of the monitor, making it essentially invisible.)

There are actually two problems with this code. First, the reliance on _SCREEN assumed the form exists within _SCREEN; with a top-level form or one with Desktop set to .T., that's not necessarily the case. Second, even if _SCREEN is maximized, that only fits it in the current monitor. If the form is on the other monitor, _SCREEN's dimensions are irrelevant.

I initially changed the code to:

```
if Thisform.Desktop or Thisform.ShowWindow = 2
    lnWidth  = sysmetric(1)
    lnHeight = sysmetric(2)
else
    lnWidth  = _screen.Width
    lnHeight = _screen.Height
endif Thisform.Desktop ...
Thisform.Width = min(max(Thisform.Width, ;
    0, Thisform.MinWidth), lnWidth)
Thisform.Height = min(max(Thisform.Height, ;
    0, Thisform.MinHeight), lnHeight)
Thisform.Left = min(max(Thisform.Left, 0), ;
    lnWidth  - 50)
Thisform.Top = min(max(Thisform.Top, 0), ;
    lnHeight - 50)
```

However, it turns out that SYSMETRIC() only returns values for the primary monitor. So, I created a tool for handling multiple monitors.

There are actually two classes, both of which are in SFMonitors.prg: SFSize, which simply has properties that represent the dimensions of a monitor, and SFMonitors, which does the work. SFMonitors is actually a subclass of SFSize because it uses those same properties for the virtual desktop (all combined monitors if there's more than one).

Here's the code for SFSize:

```
define class SFSize as Custom
    nLeft   = -1
    nRight  = -1
    nTop    = -1
    nBottom = -1
    nWidth  = 0
    nHeight = 0

    function nLeft_Assign(tnValue)
        This.nLeft = tnValue
        This.SetWidth()
    endfunc

    function nRight_Assign(tnValue)
        This.nRight = tnValue
        This.SetWidth()
    endfunc

    function nTop_Assign(tnValue)
        This.nTop = tnValue
        This.SetHeight()
    endfunc
```

```
    function nBottom_Assign(tnValue)
        This.nBottom = tnValue
        This.SetHeight()
    endfunc

    function SetWidth
        with This
            .nWidth  = .nRight - .nLeft
        endwith
    endfunc

    function SetHeight
        with This
            .nHeight = .nBottom - .nTop
        endwith
    endfunc
enddefine
```

SFMonitors has several methods. Init sets up the Windows API functions we need, determines how many monitors there are, and gets the dimensions for the primary monitor if there's only one or the virtual desktop if there's more than one. (Note: this and the other methods discussed use some constants, such as SM_CMONITORS, which are defined at the start of the PRG.)

```
function Init
  local loSize

* Declare the Windows API functions we'll
* need.

  declare integer MonitorFromPoint ;
    in Win32API ;
    long x, long y, integer dwFlags
  declare integer GetMonitorInfo ;
    in Win32API ;
    integer hMonitor, string @lpmi
  declare integer SystemParametersInfo ;
    in Win32API ;
    integer uiAction, ;
    integer uiParam, string @pvParam, ;
    integer fWinIni
  declare integer GetSystemMetrics ;
    in Win32API integer nIndex

* Determine how many monitors there are. If
* there's only one, get its size. If there's
* more than one, get the size of the virtual
* desktop.

  with This
    .nMonitors = ;
      GetSystemMetrics(SM_CMONITORS)
    if .nMonitors = 1
      loSize   = .GetPrimaryMonitorSize()
      .nRight  = loSize.nRight
      .nBottom = loSize.nBottom
      store 0 to .nLeft, .nTop
    else
      .nLeft   = ;
        GetSystemMetrics(SM_XVIRTUALSCREEN)
      .nTop    = ;
        GetSystemMetrics(SM_YVIRTUALSCREEN)
      .nRight  = ;
      GetSystemMetrics(SM_CXVIRTUALSCREEN) - ;
        abs(.nLeft)
      .nBottom = ;
      GetSystemMetrics(SM_CYVIRTUALSCREEN) - ;
        abs(.nTop)
    endif .nMonitors = 1
  endwith
endfunc
```

GetPrimaryMonitorSize returns an SFSize object for the primary monitor. Note that this takes into account the Windows Taskbar and any other desktop toolbars, which reduce the size of the available space.

```
function GetPrimaryMonitorSize
  local lcBuffer, ;
    loSize
  lcBuffer = replicate(chr(0), 16)
  SystemParametersInfo(SPI_GETWORKAREA, 0, ;
    @lcBuffer, 0)
  loSize = createobject('SFSize')
  with loSize
    .nLeft   = ctobin(substr(lcBuffer,  1, ;
      4), '4RS')
    .nTop    = ctobin(substr(lcBuffer,  5, ;
      4), '4RS')
    .nRight  = ctobin(substr(lcBuffer,  9, ;
      4), '4RS')
    .nBottom = ctobin(substr(lcBuffer, 13, ;
      4), '4RS')
  endwith
  return loSize
endfunc
```

Pass GetMonitorSize X and Y coordinates and it figures out what monitor contains that point and returns an SFSize object containing its dimensions, again accounting for the Taskbar.

```
function GetMonitorSize(tnX, tnY)
  local loSize, ;
    lhMonitor, ;
    lcBuffer
  loSize    = createobject('SFSize')
  lhMonitor = MonitorFromPoint(tnX, tnY, ;
    MONITOR_DEFAULTTONEAREST)
  if lHMonitor > 0
    lcBuffer = bintoc(40, '4RS') + ;
      replicate(chr(0), 36)
    GetMonitorInfo(lhMonitor, @lcBuffer)
    with loSize
      .nLeft   = ctobin(substr(lcBuffer, 21, ;
        4), '4RS')
      .nTop    = ctobin(substr(lcBuffer, 25, ;
        4), '4RS')
      .nRight  = ctobin(substr(lcBuffer, 29, ;
        4), '4RS')
      .nBottom = ctobin(substr(lcBuffer, 33, ;
        4), '4RS')
    endwith
  else

* Under some conditions, MonitorFromPoint
* returns a negative number, so let's use the
* primary monitor in that case.

    loSize = This.GetPrimaryMonitorSize()
  endif lHMonitor > 0
  return loSize
endfunc
```

Here's some code taken from the Restore method of SFPersistentForm (in SFPersist.vcx) that uses SFMonitors. Code before the following code (not shown here) reads a form's previous Height, Width, Top, and Left from the Windows Registry from the last time the user had it open into custom nHeight, nWidth, nTop, and nLeft properties, and then sizes and moves the form (referenced in loObject) to those values. This code

makes sure the form isn't off the screen, which can happen if, for example, the user had the form open on a second monitor but now only has one monitor, such as an undocked laptop. Note that this code uses several SYSMETRIC() functions to determine the height and width of the window border and title bar, since those values aren't included in a form's Height and Width. Also note in the comment a workaround for a peculiarity with an "in top-level form" being restored to a different monitor than the top-level form it's associated with.

```
loMonitors = newobject('SFMonitors', ;
  'SFMonitors.prg')

* For desktop or dockable forms, get the size
* of the virtual desktop. If there's only one
* monitor, use the primary monitor size.
* Otherwise, use the size of whichever monitor
* the form is on.

if pemstatus(loObject, 'Desktop', 5) and ;
  (loObject.Dockable = 1 or ;
  loObject.Desktop or loObject.ShowWindow = 2)
  if loMonitors.nMonitors = 1
    loSize = loMonitors
  else
    loSize = ;
      loMonitors.GetMonitorSize(.nLeft, .nTop)
  endif loMonitors.nMonitors = 1
  lnMaxLeft   = loSize.nLeft
  lnMaxTop    = loSize.nTop
  lnMaxWidth  = loSize.nWidth
  lnMaxHeight = loSize.nHeight
  lnMaxRight  = loSize.nRight
  lnMaxBottom = loSize.nBottom

* For any other forms, use the size of
* _screen.

else
  lnMaxLeft   = 0
  lnMaxTop    = 0
  lnMaxWidth  = _screen.Width
  lnMaxHeight = _screen.Height
  lnMaxRight  = lnMaxWidth
  lnMaxBottom = lnMaxHeight
endif pemstatus(loObject ...

* Test to see if the object is _screen.

_screen.Tag = sys(2015)
do case

* If we restored the properties, ensure the
* form isn't moved or sized outside the
* desktop boundaries. Only restore Height and
* Width if the form is resizable.

  case .WasItemRestored('Top') or ;
    .WasItemRestored('Left') or ;
    .WasItemRestored('Height') or ;
    .WasItemRestored('Width')
    llTitleBar    = pemstatus(loObject, ;
      'TitleBar', 5) and loObject.TitleBar = 1
    lnBorderStyle = ;
      icase(pemstatus(loObject, ;
      'nBorderStyle', 5), ;
      loObject.nBorderStyle, ;
      pemstatus(loObject, 'BorderStyle', 5), ;
      loObject.BorderStyle, 0)
    if lnBorderStyle = 3
      loObject.Width  = min(max(.nWidth,  0, ;
        loObject.MinWidth), lnMaxWidth)
      loObject.Height = min(max(.nHeight, 0, ;
        loObject.MinHeight), lnMaxHeight)
    endif lnBorderStyle = 3

* Calculate the total width of the form,
* including the window borders.

    if llTitleBar
      lnBorder = iif(lnBorderStyle = 3, ;
        sysmetric(3), sysmetric(12)) * 2
    else
      lnBorder = icase(lnBorderStyle = 0, 0, ;
        lnBorderStyle = 1, sysmetric(10), ;
        lnBorderStyle = 2, sysmetric(12), ;
        sysmetric(3)) * 2
    endif llTitleBar
    lnTotalWidth = loObject.Width + lnBorder
    do case

* If we're past the left edge, move it to the
* left edge.

      case .nLeft < lnMaxLeft
        loObject.Left = lnMaxLeft

* If we're past the right edge of the screen,
* move it to the right edge. We may also need
* to adjust the width to ensure it fits on the
* monitor. Only do this for a normal window;
* for maximized windows, we want to be at the
* former position.

      case ;
      .nWindowState = WINDOWSTATE_NORMAL and ;
      .nLeft + lnTotalWidth > lnMaxRight and ;
      not loObject.Tag == _screen.Tag
        loObject.Left = max(lnMaxRight - ;
          lnTotalWidth, lnMaxLeft)
        if loObject.Left + lnTotalWidth > ;
          lnMaxRight
          try
            loObject.Width = lnMaxRight - ;
              lnMaxLeft - lnBorder
          catch
          endtry
        endif loObject.Left ...

* We're cool, so put it where it was last
* time. If this form has ShowWindow set
* to 1-In Top-Level Form and the current top-
* level form is on a different monitor than
* the saved position, do this code twice; the
* first time, it gives a value that places the
* form on the wrong monitor but it works the
* second time.

      otherwise
        loObject.Left = .nLeft
        loObject.Left = .nLeft
    endcase

* Calculate the total height of the form,
* including the title bar and window borders.

    if llTitleBar
      lnVBorder = sysmetric(9) + ;
        iif(lnBorderStyle = 3, sysmetric(4), ;
        sysmetric(13)) * 2
    else
      lnVBorder = icase(lnBorderStyle = 0, ;
        0, ;
        lnBorderStyle = 1, sysmetric(11), ;
        lnBorderStyle = 2, sysmetric(13), ;
        sysmetric(4)) * 2
    endif llTitleBar
    lnTotalHeight = loObject.Height + ;
      lnVBorder
    do case

* If we're past the top edge, move it to the
```

```
* top edge.

      case .nTop < lnMaxTop
        loObject.Top = lnMaxTop

* If we're past the bottom edge of the screen,
* move it to the bottom edge. Note that we
* have to account for the height of the title
* bar and top and bottom window frame. Only do
* this for a normal window; for maximized
* windows, we want to be at the former
* position.

      case ;
    .nWindowState = WINDOWSTATE_NORMAL and ;
    .nTop + lnTotalHeight > lnMaxBottom and ;
    not loObject.Tag == _screen.Tag
        loObject.Top = max(lnMaxBottom - ;
          lnTotalHeight, lnMaxTop)
        if loObject.Top + lnTotalHeight > ;
          lnMaxBottom
          try
            loObject.Height = lnMaxBottom - ;
              lnMaxTop - lnVBorder
          catch
          endtry
        endif loObject.Top ...

* We're cool, so put it where it was last
* time.

      otherwise
        loObject.Top = .nTop
    endcase

* Bind the window's Activate event to our
* SetWindowState method; we have to set
* the WindowState once the form is visible or
* it won't be maximized on the correct
* monitor.

    if .WasItemRestored('WindowState') and ;
      .nWindowState <> WINDOWSTATE_MINIMIZED
      bindevent(loObject, 'Activate', This, ;
        'SetWindowState')
    endif .WasItemRestored('WindowState') ...

* If we didn't, force the form to AutoCenter
* if we're supposed to.

  otherwise
    loObject.AutoCenter = loObject.AutoCenter
endcase
```

Trying it out

To test how this works, try the following:

- Run the Test form (Test.scx is included with this article's downloads). This form has an SFPersistentForm object on it with cKey set to "Software\FoxRockX\TestForm", meaning its size and position is stored in the Windows Registry in HKEY_CURRENT_USER\Software\FoxRockX\TestForm.
- Size it and move it somewhere, then close it.
- Run the form again. Notice it opens at the same size and position as it was when you closed it.
- Move the form onto another monitor (because it has Desktop = .T., you can move it outside _SCREEN), then close it.
- Run the form again. Again notice it opens where it was last time. Close it.
- Unplug the monitor, then run the form again. This time it comes up on your primary monitor.

To use this in your own applications, drop an SFPersistentForm object on a form and set its cKey property to the key under HKEY_CURRENT_USER in the Registry where you want its values saved. Note you'll have to add SFCtrls.vcx, SFRegistry.vcx, SFPersist.vcx, and SFMonitors.prg to your project.

Summary

Now that you see how easy it is to persist form size and location, you'll likely become as annoyed as I am with applications that don't do this. These classes make it simple to add this capability to any of your forms.

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 (the latest being "What's New in Nine"), "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." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

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://vfpx.codeplex.com). He was a Microsoft Most Valuable Professional (MVP) from 1996 to 2011. Doug was awarded the 2006 FoxPro

Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).