Visual Studio Magazine Online
Classic VB Corner

Working with Multiple Monitors

ONLINE ONLY

Most of the time it doesn't matter where the user puts your application's windows, but every now and then you really need to know -- or even decide this for yourself.

March 2, 2009 · by Karl E. Peterson

Multiple monitor setups are becoming increasingly common, and there are occasions where you'll want to react differently based on which monitor the user places your application's windows, or even where you'll want to make sure your application is using specific monitor(s). For instance, I recently saw a newsgroup post that read in part:

> *I am writing a VBA program and need to determine the screen resolution of all the monitors in a 3-screen system. In addition, I need to know if PowerPoint is on screen one, two or three.*

I've also created applications (OK, a screensaver or three) that needed to cover the entire desktop, all visible monitors included. You've probably seen similar applications. One of my screensavers shows the same image on all monitors, while another shows different images on each monitor. In order to do this, or tasks such as the writer above describes, you need to be able to determine how many monitors there are and the screen coordinates of each. Knowing the entire desktop coordinates can also be extremely useful if you just want to cover the whole thing with a single form.

So, why not get right to that first? Multiple-monitor support wasn't offered until Windows 98 and Windows 2000, so if you're still supporting Windows 95 and/or NT4 you'll always need to code at least two solutions for every situation you intend to address. The GetSystemMetrics API call has been expanded over the years to include new built-in functionality as it increases. To determine the number of monitors present, you need only call that API, asking for SM_CMONITORS. On systems that don't support this request (there's that pesky Windows 95 again), you'll get back 0, so the best strategy is to test for >1 to quickly determine your ensuing strategy.

GetSystemMetrics provides the easiest way to retrieve the entire desktop surface coordinates. On single-monitor systems, you can assume the point 0,0 to be in the upper-left, and SM_CXSCREEN, SM_CYSCREEN to be in the lower-right. With multi-monitor systems, you need to make four GetSystemMetrics calls, one for each X and each Y value. In the case of a screensaver, you likely also want to make your canvas topmost, so the combined set of calls looks something like this:

```
' Only go into topmost mode if compiled, or
' else there is no way to debug!
If Compiled() Then
   TopMostFlag =  HWND_TOPMOST
Else
```

```
   TopMostFlag =  HWND_NOTOPMOST
End If

' Branch based on number of display monitors.
If GetSystemMetrics(SM_CMONITORS) > 1 Then
   Call  SetWindowPos(hWnd, TopMostFlag, _
                      GetSystemMetrics(SM_XVIRTUALSCREEN), _
                      GetSystemMetrics(SM_YVIRTUALSCREEN), _
                      GetSystemMetrics(SM_CXVIRTUALSCREEN), _
                      GetSystemMetrics(SM_CYVIRTUALSCREEN), _
                      SWP_SHOWWINDOW)
Else
   Call  SetWindowPos(hWnd, TopMostFlag, _
                      0, 0, _
                      GetSystemMetrics(SM_CXSCREEN), _
                      GetSystemMetrics(SM_CYSCREEN), _
                      SWP_SHOWWINDOW)
End If
```

If you need to find out information about a specific monitor, and you can point to it via a window handle or a discrete point or rectangle, there are three API functions that can be useful: MonitorFromPoint, MonitorFromRect and MonitorFromWindow (I showed how to use the latter in an earlier column). Each of these functions returns an hMonitor, which may be passed to GetMonitorInfo for overall screen and work area coordinates.

The situation becomes more complicated if you want to retrieve coordinates for all available monitors in the system. In that case, you need to set up an EnumDisplayMonitors callback. If you're simply interested in discovering monitor dimensions and positions, pass Null for all but the third parameter, which is a pointer to the callback function:

```
' Initiate  enumeration of all available displays.
Call  EnumDisplayMonitors(0&, ByVal 0&, _
                          AddressOf MonitorEnumProc, 0&)
```

Windows will then call your MonitorEnumProc function once for each monitor present, and within this procedure you can call GetMonitorInfo to gather a MONITORINFOEX structure full of information about the given monitor. Given the nature of AddressOf, you'll need to house this callback in a standard BAS module. I've written up a small sample (Monitors), which you can download from my Web site, that uses this callback to build a global collection of CMonitor objects that describe each of the devices present.

The CMonitor class exposes most of the information available for any given monitor device. In my design, it instantiates to the provide information for the entire virtual desktop, and then collects information for a specific device when passed a new Handle value. Preliminary information is gathered with GetMonitorInfo. Then, another call to

EnumDisplayDevice is made to fill a DISPLAY_DEVICE structure with additional information.

The scenario that's not often discussed is that while GetSystemMetrics(SM_CMONITORS) returns the actual number of physical monitors connected, EnumDisplayMonitors fires callbacks for both real and pseudo display devices. These phantom-like devices are created by applications like NetMeeting, and are often invisible. The best way to put it is, they're best ignored if you don't have a good reason not to.

I've found that the two best indicators that I'm dealing with a pseudo device and that I should just ignore it are found in the DISPLAY_DEVICE.dwStateFlag element. There, you'll find flags indicating whether the device is attached to the desktop and if it's a mirroring driver (which can be considered invisible).

In my CMonitor class, the Refresh method is called on instantiation whenever a new Handle is assigned or at client discretion:

```
Public Sub Refresh()
  ' Clear cached  information store.
  Call  ZeroMemory(m_MonitorInfo, Len(m_MonitorInfo))
  Call  ZeroMemory(m_DeviceInfo, Len(m_DeviceInfo))

  With  m_MonitorInfo
     ' Use all  accessible information if running under Win95/NT4.
     If  m_MultiSupport = False Then
         .rcMonitor.Right = GetSystemMetrics(SM_CXSCREEN)
         .rcMonitor.Bottom = GetSystemMetrics(SM_CYSCREEN)
         Call  SystemParametersInfo(SPI_GETWORKAREA, 0, .rcWork, 0)

     ' Fill in  structure for virtual monitor if handle is zero.
     ' No device  info associated with virtual screens.
     ElseIf  m_hMonitor = 0 Then
        With  .rcMonitor
           .Left =  GetSystemMetrics(SM_XVIRTUALSCREEN)
           .Top = GetSystemMetrics(SM_YVIRTUALSCREEN)
           .Right =  .Left + GetSystemMetrics(SM_CXVIRTUALSCREEN)
           .Bottom  = .Top + GetSystemMetrics(SM_CYVIRTUALSCREEN)
        End With
        Call  SetVirtualAttributes

     ' Otherwise,  use the full MultiMonitor API for info.
     Else
        ' Retrieve  information about this display.
        .cbSize =  Len(m_MonitorInfo)
        Call  GetMonitorInfo(m_hMonitor, m_MonitorInfo)

        ' Retrieve  information about this device.
         m_DeviceInfo.cbSize = Len(m_DeviceInfo)
```

```
        Call  EnumDisplayDevices(.szDevice(0), 0&, _
                                 m_DeviceInfo,  0&)
     End If
  End With
End Sub
```

Where m_MonitorInfo and m_DeviceInfo are module-level variables that represent the relevant API structures:

```
Private Type MONITORINFOEX
   cbSize As Long
   rcMonitor As RECT
   rcWork As RECT
   dwFlags As Long
   szDevice(0 To  CCHDEVICENAME - 1) As Byte
End Type

Private Type DISPLAY_DEVICE
   cbSize As Long
   szDevice(0 To  CCHDEVICENAME - 1) As Byte
   szDeviceString(0  To 127) As Byte
   dwStateFlags As  Long
   szDeviceID(0 To  127) As Byte
   szDeviceKey(0 To  127) As Byte
End Type

Private m_MonitorInfo As MONITORINFOEX
Private m_DeviceInfo As DISPLAY_DEVICE
```

In my design, the collection of CMonitor objects is keyed in the order the system identifies them in the Display Properties dialog Settings tab. So, if you wanted to query monitor 1:

```
Debug.Print Monitors("1").Primary
```

I use the Key of 0 (zero) for the virtual screen, so if, say, you wanted your form to cover the entire screen, you could do this:

```
With Monitors("0")
   .PixelsToTwips =  True
   Me.Move .Left,  .Top, .Width, .Height
End With
```

Because VB collections are 1-based, the Index properties are all one off, so you'll need to be aware of that. Each CMonitor object exposes an Index property (Long) that replicates the Key used when it was added to the collection. I find this easier to keep straight in my mind, if it's all 0-based, and this matches how Windows identifies the monitors as well.

It takes awhile to become familiar with the vagaries of multi-monitor setups. For a good background read, I'd recommend this old MSJ article. Be aware that the coordinates can be, and often are, negative -- it's not at all uncommon for 0,0 to be in the middle of the virtual desktop. Finally, swing by my Web site and grab the Monitors sample if you'd like to get started playing with this capability right away.

I'm curious, too, if you're writing new ClassicVB apps -- do you explicitly support or disavow support for "antique" operating systems, or just sort of close your eyes and hope none of your users are still using those? In particular, do you go out of your way to support NT4 and/or Windows 95? Leave me some feedback, here or at my web site. Thanks!

Oh, one more link to the sample code...

About the Author

Karl E. Peterson wrote Q&A, Programming Techniques, and various other columns for VBPJ and VSM from 1995 onward, until Classic VB columns were dropped entirely in favor of other languages. Similarly, Karl was a Microsoft BASIC MVP from 1994 through 2005, until such community contributions were no longer deemed valuable. He is the author of VisualStudioMagazine.com's new Classic VB Corner column. You can contact him through his Web site if you'd like to suggest future topics for this column.