July, 2007

Print Forms as Users See Them

Using the GDIPlusX classes and some ingenuity, you can even print scrollable forms.

By Tamar E. Granor, technical editor

I'm working on an application where I'm building complex forms out of a variety of container, shape, and line objects. The forms are a graphical representation of real-world objects and users can design the objects in question by adding and removing items from the forms and moving things around. They need the ability to print what they've designed.

What doesn't work

I considered several solutions to this problem. My first thought was to use Windows' built-in PrintScreen ability, but it can't print a single VFP form. You can use it to grab the entire Windows desktop (PrintScrn) or just the current application (Alt-PrintScrn), but there's no way to tell it to grab just a form.

Next, I considered using a third-party tool, like SnagIt, that can be automated; but my client didn't want to pay for an add-on for every installation.

My next thought was to create a report to replicate the screen. But I wasn't enthusiastic about this approach because I knew how much work went into building the forms and anticipated a similar complexity level.

Turning to GDIPlusX

At that point, fortunately, I realized that the GDIPlusX project that's part of VFPX might offer an alternative way to build a report. As I looked through the examples that come with GDIPlusX, I found one to convert a form into a bitmap. Not only was this exactly what I needed, but it required only a few lines of code.

GDI+ is a DLL that centralizes graphics-handling in much the same way that the Windows printing system centralizes output. See Christof's article in the July 2004 issue for an explanation of GDI+ and how it works (http://My.Advisor.com/doc/XXXXX).

GDIPlusX is a set of class libraries designed to simplify integrating GDI+ into applications. The objects and methods are modeled after the GDI+ classes in the .NET framework (the System.Drawing namespace) and the developers of GDIPlusX have even done the work to let us refer to objects in the same way you would when working with System.Drawing.

I added a PrintObject method to my form base class. (I couldn't call it Print, because that's a built-in form method.) To use any of the GDIPlusX classes, you need to instantiate the xfcSystem class found in System.VCX. The examples that come with GDIPlusX add a property to the _Screen object for this. I chose instead to add a form-level property to hold this reference and populated it in the form's Load method, so it would be available throughout the form:

```
This.oGDIPlusSystem = NEWOBJECT("xfcSystem","system.vcx")
```

In the PrintObject method, the code to capture the screen and save it to a bitmap file is quite simple:

```
cFileName = FORCEEXT(FORCEPATH(This.Name, SYS(2023)), ;
                     "BMP")
CLEAR RESOURCES (cFileName)
loCaptureBmp = ;
  This.oGDIPlusSystem.Drawing.Bitmap.FromScreen( ;
    Thisform.HWnd)
loCaptureBmp.Save(cFileName, ;
  This.oGDIPlusSystem.Drawing.Imaging.ImageFormat.BMP)
```

The first line creates a filename by adding a path and a BMP extension to the form's Name. I considered using the form's Caption instead, but the captions of these forms can include some characters that aren't permitted in file names. The filename doesn't really matter because the file is deleted at the end of the method.

VFP likes to cache things, including graphical objects. The CLEAR RESOURCES command ensures that the code that prints the bitmap later in this method doesn't reuse a previous version of the bitmap.

The last two lines here capture and save the form. The System.Drawing.Bitmap.FromScreen method creates a bitmap object from the form whose hWnd is passed in. The Save method of the bitmap object saves the image in the specified file and format. Note that you can actually save in a variety of formats, by changing the second parameter passed to Save. The method supports a number of other formats, include JPEG, TIFF, and GIF.

Printing the image

Getting the bitmap printed turned out to be a little more complicated than I expected. My original plan was to use the ShellExecute API function (through the _ShellExecute class in the FoxPro Foundation classes). However, my tests revealed that, on my machine, there was no Print action defined for bitmap files, nor for any of the other common graphic formats. If this Print action wasn't defined on my machine, it probably wouldn't be on the user's machines either.

I found an alternative approach on Cesar Chalom's blog at http://weblogs.foxite.com/cesarchalom/archive/2007/01/17/3143.aspx. (Incidentally, Cesar is one of the principal contributors to GDIPlusX.) The basic idea is to create a report programmatically and add an image object to it, pointing that object to the file you want to print. Then run the report and, finally, delete it. Cesar's code, which is adapted from a Microsoft Knowledge Base article, uses two methods. Because I needed this only for a very specific purpose, I flattened Cesar's code and put it all within the PrintObject method.

In my application, users can resize the forms. So I didn't know in advance whether a given form would print better in portrait mode or landscape mode. But I wanted to use whichever made the most sense for a given form. I added this line to figure out which mode to use:

```
lPortrait = (This.Height > This.Width)
```

Then, I was able to use the lPortrait variable to configure the report to use the appropriate orientation. Here's my version of the reporting code:

```
LOCAL lnArea
lnArea = SELECT()
CREATE CURSOR ReportTemp (ImageFile c(150))
INSERT INTO ReportTemp VALUES (m.cFileName)
CREATE REPORT ___ImageReport FROM ReportTemp
*-- Open the report file (FRX) as a table.
USE "___ImageReport.FRX" IN 0 ALIAS TheReport EXCLUSIVE
SELECT TheReport
*-- Remove from the FRX the auto generated fields
*-- and labels
DELETE FROM TheReport WHERE ObjType = 5 AND ;
                            ObjCode = 0
DELETE FROM TheReport WHERE ObjType = 8 AND ;
                            ObjCode = 0
*-- Find the header record (normally the first one,
*-- but who knows)
LOCATE FOR ObjType = 1 AND ObjCode = 53
IF FOUND()
```

```
   REPLACE Tag WITH "",; && Clears the Tag, Tag2
      Tag2 WITH ""

   IF NOT m.lPortrait
      REPLACE Expr WITH "ORIENTATION=1"
   ENDIF
ENDIF

*-- Add a Picture/OLE Bound control to the report by
*-- inserting a record with appropriate values. Using
*-- an object that is based on the EMPTY class here
*-- and the GATHER NAME class later to insert the
*-- record makes it easier to see which values line
*-- up to which fields (when compared to a large
*-- SQL-INSERT command).
LOCAL loNewRecObj AS EMPTY
loNewRecObj = NEWOBJECT( 'EMPTY' )
ADDPROPERTY( loNewRecObj, 'PLATFORM', 'WINDOWS' )
ADDPROPERTY( loNewRecObj, 'Uniqueid', SYS(2015) )
ADDPROPERTY( loNewRecObj, 'ObjType', 17 )
  && "Picture/OLE Bound Control"
ADDPROPERTY( loNewRecObj, 'NAME', ;
            'ReportTemp.ImageFile' )
  && The object ref to the IMAGE object.
ADDPROPERTY( loNewRecObj, 'Hpos', 100)
ADDPROPERTY( loNewRecObj, 'Vpos', 600)
ADDPROPERTY( loNewRecObj, 'HEIGHT', 100000)
ADDPROPERTY( loNewRecObj, 'WIDTH', 100000)
ADDPROPERTY( loNewRecObj, 'DOUBLE', .T. )
  && Picture is centered in the control
ADDPROPERTY( loNewRecObj, 'Supalways', .T. )
*-- For the Picture/OLE Bound control, the contents
*-- of the OFFSET field specify whether Filename (0),
*-- General field name (1), or Expression (2)
*-- is the source.
ADDPROPERTY( loNewRecObj, 'Offset', 2 )
*-- Add the Picture/OLE Bound control record.
APPEND BLANK IN TheReport
GATHER NAME loNewRecObj MEMO
*-- Clean up and then close the report table.
PACK
USE IN SELECT('TheReport')
*-- Make sure that the cursor is selected,
*-- and then run the report to preview using
*-- the instance of our Report Listener.
SELECT ReportTemp
REPORT FORM "___ImageReport" TO PRINT
DELETE FILE "___ImageReport.fr*"
SELECT (lnArea)
```

Dealing with scrolling

At this point, I would have been done, except for one thing. The forms in question are scrollable and the users want to see everything on the form, not just the part that's currently displayed on screen.

I needed to resize the forms so that everything on the form was visible. This was complicated by the fact that the forms use anchoring to keep controls where they should be, and one of the forms enlarges and shrinks objects when the form is resized, rather than showing more objects. A form property, lChangeFontOnResize, controls this behavior.

To handle all these issues, I added two methods, ShowAll and Restore, that enlarge the form to make all its objects visible and restore it to its previous size, respectively.

ShowAll saves the current height and width in custom form properties, then calls another method, FindLargest, to determine the height and width required to show all objects on the form. Then, if the form changes object and font sizes when resized, that feature and anchoring on the form are turned off. Then the form is resized. Here's the code for ShowAll:

```
LOCAL oObject, nMaxRight, nMaxBottom, nAnchor
nMaxRight = This.Width
nMaxBottom = This.Height
This.nHoldWidth = This.Width
This.nHoldHeight = This.Height
This.lHoldChangeFont = This.lChangeFontOnResize
IF This.ScrollBars <> 0
  This.FindLargest(This, @nMaxRight, @nMaxBottom)
  IF m.nMaxRight > This.Width OR ;
     m.nMaxBottom > This.Height
    IF This.lChangeFontOnResize
      This.lChangeFontOnResize = .F.
      * Need to turn off anchoring so resize works
      This.TurnOffAnchors()
    ENDIF
    * Adjust stored value to account for scrollbars
    IF m.nMaxRight > This.Width
      This.nHoldHeight = This.nHoldHeight + SYSMETRIC(14)
    ENDIF

    IF m.nMaxBottom > This.Height
      This.nHoldWidth = This.nHoldWidth + SYSMETRIC(15)
    ENDIF
    This.Width = m.nMaxRight
    This.Height = m.nMaxBottom
```

```
    * Let drawing catch up
    DOEVENTS FORCE
  ENDIF
ENDIF
RETURN
```

The DOEVENTS command lets the form finish resizing and redrawing before the capture.

FindLargest is a simple method that traverses the object hierarchy of the form, finding the right-most and bottom-most positions for any objects:

```
* Find the objects at the extreme of the form
LPARAMETERS nMaxRight, nMaxBottom
LOCAL oObject
FOR EACH oObject IN This.Objects
  IF PEMSTATUS(oObject, "Left", 5) AND ;
     PEMSTATUS(oObject, "Width", 5)
    IF oObject.Left + oObject.Width > m.nMaxRight
      nMaxRight = oObject.Left + oObject.Width
    ENDIF
  ENDIF
  IF PEMSTATUS(oObject, "Top", 5) AND ;
     PEMSTATUS(oObject, "Height", 5)
    IF oObject.Top + oObject.Height > m.nMaxBottom
      nMaxBottom = oObject.Top + oObject.Height
    ENDIF
  ENDIF
ENDFOR
RETURN
```

The custom TurnOffAnchors method goes through the top-level objects on the form, stores their Anchor properties in a collection so you can restore them later, and sets Anchor to 0 for each. It's sufficient to do this only at the top level since anchoring won't kick in for contained objects when the container has anchoring turned off.

```
LOCAL oObject, oAnchorData
This.oAnchors = CREATEOBJECT("Collection")
FOR EACH oObject IN This.Objects
  IF PEMSTATUS(oObject, "Anchor", 5) AND ;
     oObject.Anchor <> 0
    oAnchorData = CREATEOBJECT("Empty")
    ADDPROPERTY(oAnchorData, "oObject", m.oObject)
    ADDPROPERTY(oAnchorData, "nAnchor", ;
                oObject.Anchor)
    This.oAnchors.Add(oAnchorData)

    oObject.Anchor = 0
  ENDIF
ENDFOR
```

```
RETURN
```

The Restore method undoes all this. It resets the form's size, and if appropriate, turns anchoring back on:

```
IF This.nHoldHeight <> This.Height OR ;
   This.nHoldWidth <> This.Width

  This.Height = This.nHoldHeight
  This.Width = This.nHoldWidth

  IF This.lHoldChangeFont
    This.TurnOnAnchors()
    This.lChangeFontOnResize = This.lHoldChangeFont
  ENDIF
ENDIF
```

TurnOnAnchors simply spins through the collection created in TurnOffAnchors and restores the original values:

```
LOCAL oAnchor
FOR EACH oAnchor IN This.oAnchors
  oObject = oAnchor.oObject
  oObject.Anchor = oAnchor.nAnchor
ENDFOR
This.oAnchors = .null.
```

I modified PrintObject to call ShowAll before capturing the form, and to call Restore following the capture:

```
This.ShowAll()
loCaptureBmp = ;
  This.oGDIPlusSystem.Drawing.Bitmap.FromScreen( ;
    Thisform.HWnd)
loCaptureBmp.Save(cFileName, ;
  This.oGDIPlusSystem.Drawing.Imaging.ImageFormat.BMP)
lPortrait = (This.Height > This.Width)
This.Restore()
```

There's one final issue. When the code runs, there's a "flash" as the form is resized and then restored. I tried using LockScreen to prevent the flash, but that defeated the purpose of resizing, since the FromScreen method takes a "picture" of what it actually sees. Ultimately, I decided that the users would interpret the flash as something happening.

That's all, folks

By adding PrintObject to my base form class, every form in this application can print itself. Thanks to the efforts of the GDIPlusX team, what could have been a really difficult task turned out to be reasonably

straightforward, and I got a reminder of the value of looking at other people's code.

A form class incorporating this technique is included on this month's Professional Resource CD. You'll need to download and unzip the GDIPlusX libraries (from http://www.codeplex.com/VFPX/Wiki/View.aspx?title=GDIPlusX) and set an appropriate path before using the printing functionality.