**WPF Visual Print Component**

By [**Fred Song (Melbourne)**](http://www.codeproject.com/script/Membership/View.aspx?mid=3827043) | 11 Aug 2011

[C#](http://www.codeproject.com/search.aspx?aidlst=81)[.NET](http://www.codeproject.com/search.aspx?aidlst=98)[Dev](http://www.codeproject.com/search.aspx?aidlst=118)[WPF](http://www.codeproject.com/search.aspx?aidlst=140)[Intermediate](http://www.codeproject.com/search.aspx?aidlst=153)

A look at how to print a visual element with the WPF Visual Print

* [Download source code - 636.28 KB](http://www.codeproject.com/KB/WPF/WPF_Visual_Print/WPF_Visual_Print.zip)
* [Download VisualPrint update - 15.18 KB](http://www.codeproject.com/KB/WPF/WPF_Visual_Print/VisualPrint.zip)

**Introduction**

Often we need to print the screen or some parts of the screen. This is pretty useful for printing graphic reports, like charts. In this article, we’re going to have a look at how to print a visual element with the WPF Visual Print. The essential part is serializing an XAML element to an XPS document, and converting the XPS document to a FlowDocument. Then we print and print preview the FlowDocument with the FlowDocument Viewer.

**What is an XPS Document**

The XML Paper Specification (XPS) format is basically an electronic representation of digital documents based on XML. It is a paginated fixed-layout format that retains the look and feel of your electronic documents. XPS documents can be easily created once you have the right software installed, like Microsoft Word.

The parts of an XPS document are organized in a logical hierarchy with the FixedDocumentSequence part at the top. An XPS document package may contain more than one document, and the sequence of these documents is described by the FixedDocumentSequence part. The FixedDocumentSequence part references the FixedDocument parts that, in turn, reference the pages of each document within the package.

Each FixedDocument part references the pages of that document as FixedPage parts. Each FixedPage part contains the text markup and layout of a page in the document as well as references to images, fonts, and other custom resources used in the page. Resources such as images and fonts are stored in the package but outside of the FixedPage part, allowing them to be shared by other pages. This is especially useful for font resources, but it could also be useful for any image resource that is used on more than one page, such as a watermark or letterhead logo.

I know it’s pretty boring for you to read these definitions. But I have to bring them out, because all these definitions will be used in the WPF Visual Print code.

**Serialize a Visual Component to XPS Document**

XPS documents are stored in a file, called a package, that conforms to the Open Packaging Conventions and are composed of a set of document components known as parts. A package has a physical and a logical organization. The physical organization consists of the document parts and folders inside the package, and the logical organization is a hierarchy described by the document parts. The XML Paper Specification applies a specific organization and naming convention to the logical layer for XPS documents.

WPF wraps the XPS API in the XPSSerializationManager class. Thus WPF can create an XPS document by serializing the XAML element page to an XPS document. Here is our serialization code:

FrameworkElement fe = (visual as FrameworkElement);

fe.Measure(new Size(Int32.MaxValue, Int32.MaxValue));

Size visualSize = fe.DesiredSize;

fe.Arrange(new Rect(new Point(0, 0), visualSize));

MemoryStream stream = new MemoryStream();

string pack = "pack://temp.xps";

Uri uri = new Uri(pack);

DocumentPaginator paginator;

XpsDocument xpsDoc;

using (Package container = Package.Open(stream, FileMode.Create))

{

PackageStore.AddPackage(uri, container);

using (xpsDoc = new XpsDocument(container, CompressionOption.Fast, pack))

{

XpsSerializationManager rsm =

new XpsSerializationManager(new XpsPackagingPolicy(xpsDoc), false);

rsm.SaveAsXaml(visual);

paginator = ((IDocumentPaginatorSource)

xpsDoc.GetFixedDocumentSequence()).DocumentPaginator;

paginator.PageSize = visualSize;

}

PackageStore.RemovePackage(uri);

}

**Custom Document**

When you serialize an XAML element to an XPS document, it’s always one page. Apparently, it’s not good enough for visual elements that need multiple pages. So you need to write your own paginator class, VisualDocumentPaginator.

The VisualDocumentPaginator constructor modifies the page size of the original paginator based on the required page size and margin. The new GetPage method calls the original GetPage method to get a page, then tries to measure the visual element size and split it to multiple pages per page size. Here is our VisualDocumentPaginator class:

public class VisualDocumentPaginator : DocumentPaginator

{

Size m\_PageSize;

Size m\_Margin;

DocumentPaginator m\_Paginator = null;

int m\_PageCount;

Size m\_ContentSize;

ContainerVisual m\_PageContent;

ContainerVisual m\_SmallerPage;

ContainerVisual m\_SmallerPageContainer;

ContainerVisual m\_NewPage;

public VisualDocumentPaginator(DocumentPaginator paginator,

Size pageSize, Size margin)

{

m\_PageSize = pageSize;

m\_Margin = margin;

m\_Paginator = paginator;

m\_ContentSize = new Size(pageSize.Width - 2 \* margin.Width,

pageSize.Height - 2 \* margin.Height);

m\_PageCount = (int)Math.Ceiling(m\_Paginator.PageSize.Height /

m\_ContentSize.Height);

m\_Paginator.PageSize = m\_ContentSize;

m\_PageContent = new ContainerVisual();

m\_SmallerPage = new ContainerVisual();

m\_NewPage = new ContainerVisual();

m\_SmallerPageContainer = new ContainerVisual();

}

Rect Move(Rect rect)

{

if (rect.IsEmpty)

{

return rect;

}

else

{

return new Rect(rect.Left + m\_Margin.Width,

rect.Top + m\_Margin.Height,

rect.Width, rect.Height);

}

}

public override DocumentPage GetPage(int pageNumber)

{

m\_PageContent.Children.Clear();

m\_SmallerPage.Children.Clear();

m\_NewPage.Children.Clear();

m\_SmallerPageContainer.Children.Clear();

DrawingVisual title = new DrawingVisual();

using (DrawingContext ctx = title.RenderOpen())

{

FontFamily font = new FontFamily("Times New Roman");

Typeface typeface =

new Typeface(font, FontStyles.Normal,

FontWeights.Bold, FontStretches.Normal);

FormattedText text = new FormattedText("Page " +

(pageNumber + 1) + " of " + m\_PageCount,

System.Globalization.CultureInfo.CurrentCulture,

FlowDirection.LeftToRight,

typeface, 14, Brushes.Black);

ctx.DrawText(text, new Point(0, 0));

}

DocumentPage page = m\_Paginator.GetPage(0);

m\_PageContent.Children.Add(page.Visual);

RectangleGeometry clip = new RectangleGeometry(

new Rect(0, m\_ContentSize.Height \* pageNumber,

m\_ContentSize.Width, m\_ContentSize.Height));

m\_PageContent.Clip = clip;

m\_PageContent.Transform =

new TranslateTransform(0, -m\_ContentSize.Height \* pageNumber);

m\_SmallerPage.Children.Add(m\_PageContent);

m\_SmallerPage.Transform = new ScaleTransform(0.95,0.95);

m\_SmallerPageContainer.Children.Add(m\_SmallerPage);

m\_SmallerPageContainer.Transform = new TranslateTransform(0, 24);

m\_NewPage.Children.Add(title);

m\_NewPage.Children.Add(m\_SmallerPageContainer);

m\_NewPage.Transform =

new TranslateTransform(m\_Margin.Width, m\_Margin.Height);

return new DocumentPage(m\_NewPage, m\_PageSize,

Move(page.BleedBox),Move(page.ContentBox));

}

public override bool IsPageCountValid

{

get

{

return true;

}

}

public override int PageCount>

{

get

{

return m\_PageCount;

}

}

public override Size PageSize

{

get

{

return m\_Paginator.PageSize;

}

set

{

m\_Paginator.PageSize = value;

}

}

public override IDocumentPaginatorSource Source

{

get

{

if (m\_Paginator != null)

return m\_Paginator.Source;

return null;

}

}

}

Here is the code which converts the default XPS document to a multiple pages XPS document:

using (Package container = Package.Open(stream, FileMode.Create))

{

using (xpsDoc = new XpsDocument(container, CompressionOption.Fast, pack))

{

paginator = new VisualDocumentPaginator(paginator,

new Size(pageSize.Width, pageSize.Height),

new Size(48, 48));

XpsSerializationManager rsm = new XpsSerializationManager(

new XpsPackagingPolicy(xpsDoc), false);

rsm.SaveAsXaml(paginator);

}

PackageStore.RemovePackage(uri);

}

**Print and Print Preview with FlowDocument**

**Why We Use FlowDocument**

A flow document is designed to "reflow content" depending on the window size, device resolution, and other environment variables. In addition, flow documents have a number of built-in features including search, viewing modes that optimize readability, and the ability to change the size and appearance of fonts. Flow documents are best utilized when ease of reading is the primary document consumption scenario.

**Convert an XPS Document to a FlowDocument**

There is [an excellent article](http://khason.net/blog/converting-fixeddocument-xpsdocument-too-to-flowdocument/) that talks about how to convert an XPS document to a flow document. I won’t repeat it here. As we talked before, an XPS document consists of multiple fixed pages. Every fixed page contains UI Element children. So in short words, this conversion extracts the UI Element children of a FixedPage and adds them to a FlowDocument UI block.

One thing I need to bring out is, the font resource in an XPS document is obfuscated. These fonts normally are saved as ODTTF files. Before converting to a flowdocument, we need to de-obfuscate first. In .NET Framework 3.5, we still use the same name for the de-obfuscated font file to generate a Glyph. But in .NET Framework 4.0, you’ll get a null reference exception when you try to generate a Glyph with an ODTTF file, even this file is de-obfuscated. That’s null reference exception from MS.Internal.FontCache.FontFaceLayoutInfo.IntMap.TryGetValue( Int32 key, UInt16& value)”. So you have to rename the de-obfuscated font file to a TTF file, and save font file to different folder. You can name the folder as font name.

Shown below is the code to de-obfuscate an ODTTF font. You can get the GUID from the ODTT font file name:

private static void DeobfuscateData(byte[] fontData, string guid)

{

byte[] guidBytes = new byte[16];

for (int i = 0; i < guidBytes.Length; i++)

{

guidBytes[i] = Convert.ToByte(guid.Substring(i \* 2, 2), 16);

}

for (int i = 0; i < 32; i++)

{

int gi = guidBytes.Length - (i % guidBytes.Length) - 1;

fontData[i] ^= guidBytes[gi];

}

}

**Print FlowDocument**

Call the PrintDialog.PrintDocument method to call the print dialog that will allow you to select a printer and send a document to the printer to print it. The PrintDocument method of PrintDialog takes a DocumentPaginator object that you can get from the IDocumentPaginatorSource.DocumentPaginator property as listed in the following code:

DocumentPaginator paginator =

((IDocumentPaginatorSource)m\_FlowDocument).DocumentPaginator;

m\_PrintDialog.PrintDocument(paginator, "Printing");

**License**

This article, along with any associated source code and files, is licensed under [The Code Project Open License (CPOL)](http://www.codeproject.com/info/cpol10.aspx)

**About the Author**