Duncan Jones

# Printing—a .NET developer's guide

# Printing – a .NET developers guide


Duncan@merrioncomputing.com
http://www.merrioncomputing.com

# Table of Contents

Chapter 1

# Introduction

**Who is this book for?**

This book is targeted at .NET developers who need to add printing functionality to a windows forms application, or to interact with the print subsystem. It assumes familiarity with either VB.NET or C# but no specialist printing experience is required.

**Why is it needed?**

Printing is a topic that gets very slight – if any – coverage in the majority of .NET manuals, and although there is a significant amount of documentation on MSDN for each of the common runtime classes pertaining to printing it isn't always as accessible as it could be.

This is in spite of the fact that a very significant part of application development involves printing, and a high percentage of the questions on the newsgroups and forums are printing related.

**What is included?**

The book starts with a description of the windows print subsystem with the main components identified. It then describes the .NET classes used to interact with this system to produce printed documents – both in .NET 2.0 (windows forms) and .NET 3.0 (windows presentation framework ™). It also details the file formats used in the printer spooler including the new XML paper specification (XPS) format.

The book is in two parts. Part 1 is focussed on Windows Forms and the .NET framework versions 1.0 to 3.0. Part 2 focuses on the Windows Presentation Framework and the XML paper specification.

**What is not included?**

This book does not cover third party reporting tools (such as *Crystal Reports*) as these either have their own documentation or already have reference books dedicated to them.

It does not cover print devices that do not use the core print subsystems of the windows operating system. Examples of these are barcode printers, thermal receipt printers and very large format poster printing devices.

Additionally, operating systems other than Microsoft Windows (XP, Vista, Server 2000, Server 2003 or Server 2008) are beyond the scope of this book.

#### Conventions used in this book

Source code is in `courier`, with footnotes denoting which languages or technologies are targeted.

#### Licensing and copyright

Chapter

2

# It all starts with a blank sheet of paper

## Overview of the print process

Although it is possible to live a happy and fulfilling life without ever looking under the bonnet at the print subsystem, it does make it easier to write an application that has hardcopy capabilities if you have some understanding of what the Operating System does in response to the "print" command.

To achieve this understanding it helps to look at an overview of the process from clicking the print menu to picking up the completed print job from three points of view: that of the user, that of the application doing the printing and that of the operating system elements that perform the print operation.

#### The user's perspective

When the user presses the print button or clicks on the print menu the application brings up a dialog box that lists the printers installed on the system and allows the user to select other options such as the number of copies to print, the print quality and so on.

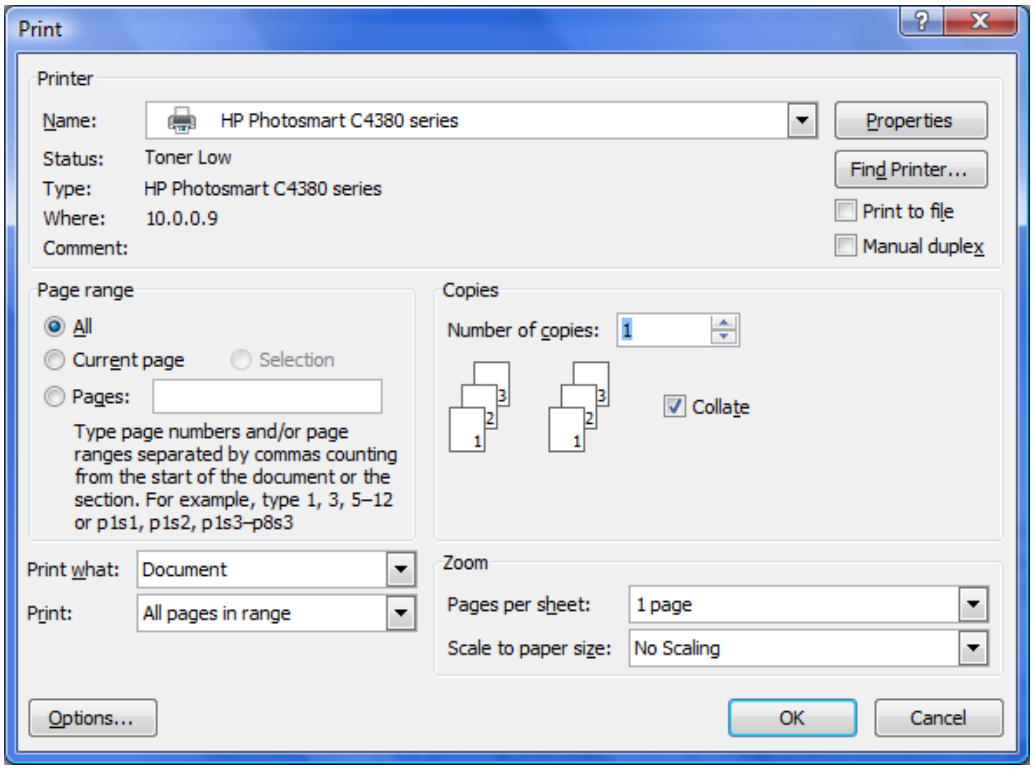


Figure 1 - Print Options dialog

The user selects the desired print settings and presses OK. At this stage the document is written to the print spooler and many applications show an on screen progress notification label such as "**Printing Page 1/7**".

There is also a notification icon in the system tray and if the user double-clicks on this the spool queue for the selected printer is displayed:

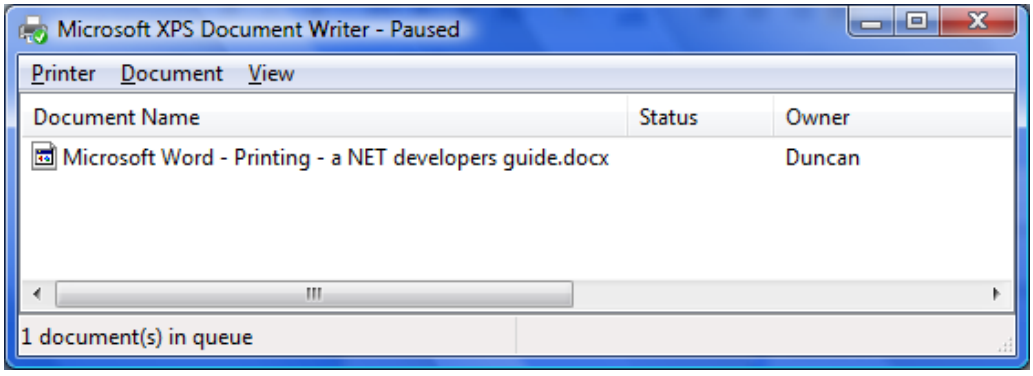


Figure 2 - The printer spool queue

Finally the printer prints out the document and the job is removed from the spool queue[1].

#### The application perspective[2]

When an application receives notification that a user wants to print a document it typically offers the user the opportunity to select the device to print on and any settings they want to apply to this print job.

The application then gets the user's print settings and notifies the print spooler that it wants to print a job, and then that it wants to print the first page. The application gets back a canvas[3] on which it can draw with the built in graphics commands to achieve the desired page. When this is done it then notifies the spooler which takes the canvas (and more importantly the graphical instructions inscribed upon it) and converts it to the language that the printer understands and writes it to the spool file. If the application has more pages to print it requests another page and the process is repeated until the application notifies the spooler that the job is completed.

#### The operating system perspective

The first interaction between the operating system and the application comes when the application requests the standard print setup dialog box. In response the operating system gets a list of all the printers installed on the system and sets the listbox to the printer that is set as the default. It then gets the current status of the printer and the current device settings and displays them on the dialog box.

When the user has set up the printer to their requirements and presses OK, the operating system passes the setup data back to the calling application.

The next interaction occurs when the application requests the operating system to start a print job. It passes a document name and the initial job settings to the operating system which creates a spool file and notifies the application that it is ready to process the print operation.

When the application requests a page to print, the operating system passes back a device context for the application to perform the drawing and text operations onto that represent a printed page, and when the application indicates that it has completed these operations it translates these drawing and text operations into the language understood by the print device and writes them to the spool file. This is repeated for each subsequent page until the application indicates that it has sent the last page.

---

[1] There is an option **Keep Printed Documents** which, when selected, means the spool file is not deleted when the job is printed. This is discussed in the section on printer settings.

[2] This describes an average application's perspective – some specialist applications have a much more involved print process.

[3] Also known as a Device Context in windows API programming

At the same time another process is reading completed pages from the spool file and sending them to the physical print device. Each time data is written to the printer the spooler gets updated as to the status of the printer and it passes this and the print job progress information on to any applications that are monitoring the printer.

## Print components

The print process as detailed above is made possible by a number of components which work together to get the application's designs down on paper.

#### Spooler service

The spooler service (spoolsrv.exe) is responsible for overseeing the process and the communication between the other components in the print process. There is only one spooler service on each server which manages all the printers on that server and all the clients connecting to it. The same spooler service is in the Windows client and performs the same role in relation to locally installed print devices.

#### Print Processor

A print processor is responsible for converting the graphics and text instructions stored in a spool file into the format that can be sent to a print monitor and also for dealing with requests to manage a print queue by pausing, resuming or cancelling jobs.

#### Printer Driver

A print driver is tied very closely to the physical hardware or software-only print device. It provides information about the capabilities of the device (such as the paper sizes supported and the maximum resolution available) and it allows the rest of the print architecture to operate without any knowledge of the internals of the print device itself.

Chapter

3

# Settings for printers and printing

There are a number of settings that can be configured using the control panel, through the printer settings dialog box or by an application. A typical printer settings dialog is as follows:

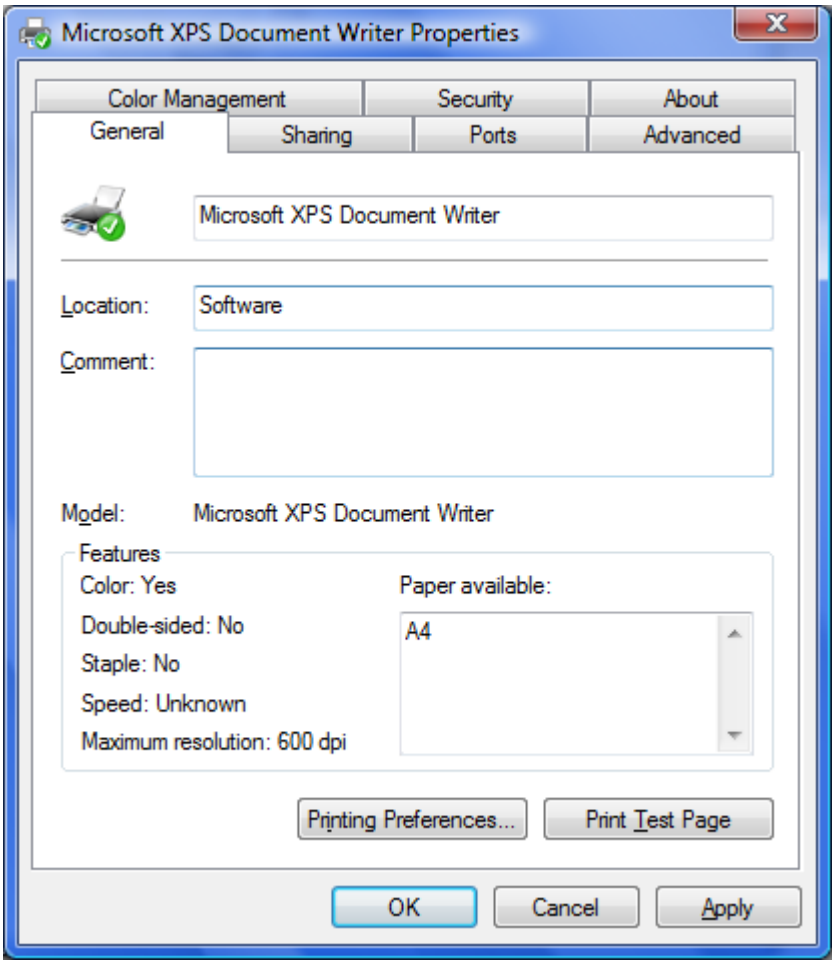


Figure 3 - Printer Properties Dialog

The dialog provides a mix of information about the print device and also settings that can be changed for that device, subject to the current user having sufficient privilege to change printer settings.

On the general tab the settings that can be changed are the *printer name*, the *comment* associated with that printer and also a *location* field that can be used to denote the physical location of the print device.

Of these the printer name is the most important setting as this name is used by the .NET framework [4]in order to identify this particular print device.

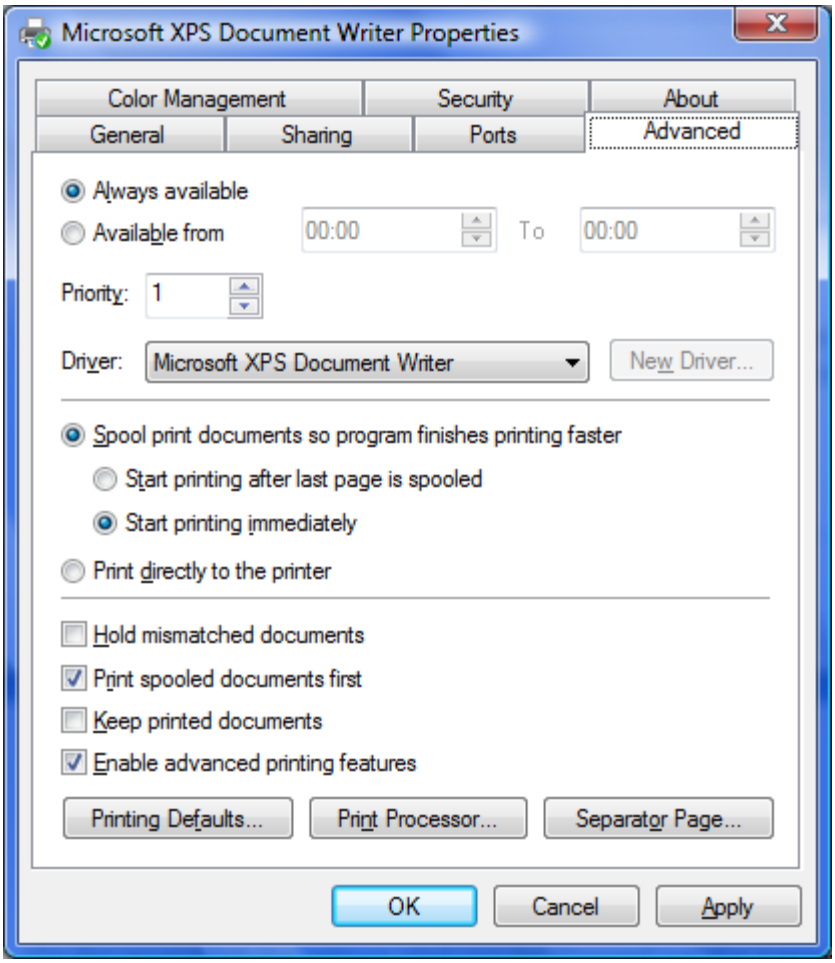


Figure 4 - Advanced printer properties

There is also a tab for advanced printer properties that allows you to set a time window in which the printer is available (this is usually used only in corporate environments), the print driver to use to print on the device, settings that control how (or if) the spooler is used in queuing jobs for the printer and additional print control options.

All of these print properties that the user can set from the control panel can also be set from a .NET application.

[4] This also applies to non-.NET applications as the device name is used by the underlying Windows API

If you are using .NET framework version 3.0 or higher this is done using the `System.Printing` namespace, but if you are using a framework prior to this you have to fall back on the Windows API[5].

[5] Subject to the permissions of the account that executes the code

Chapter

4

# Printing in .NET with windows forms

The .NET framework has a number of classes built in to allow you to print a document with a balance of as much power and as much ease of use as possible. These classes and the events they provide closely map to the print process as viewed from the application and operating system perspective in chapter 2.

#### The PrintDocument class

The `PrintDocument` class is at the heart of printing from a windows forms application. It has a single method - `PrintDocument.Print` - and when that method is called in code (usually in response to a user action such as clicking a print menu) the class raises events that allow your application to interact with the print subsystem in a process that is very similar to the application view of the print process described in Chapter 2.

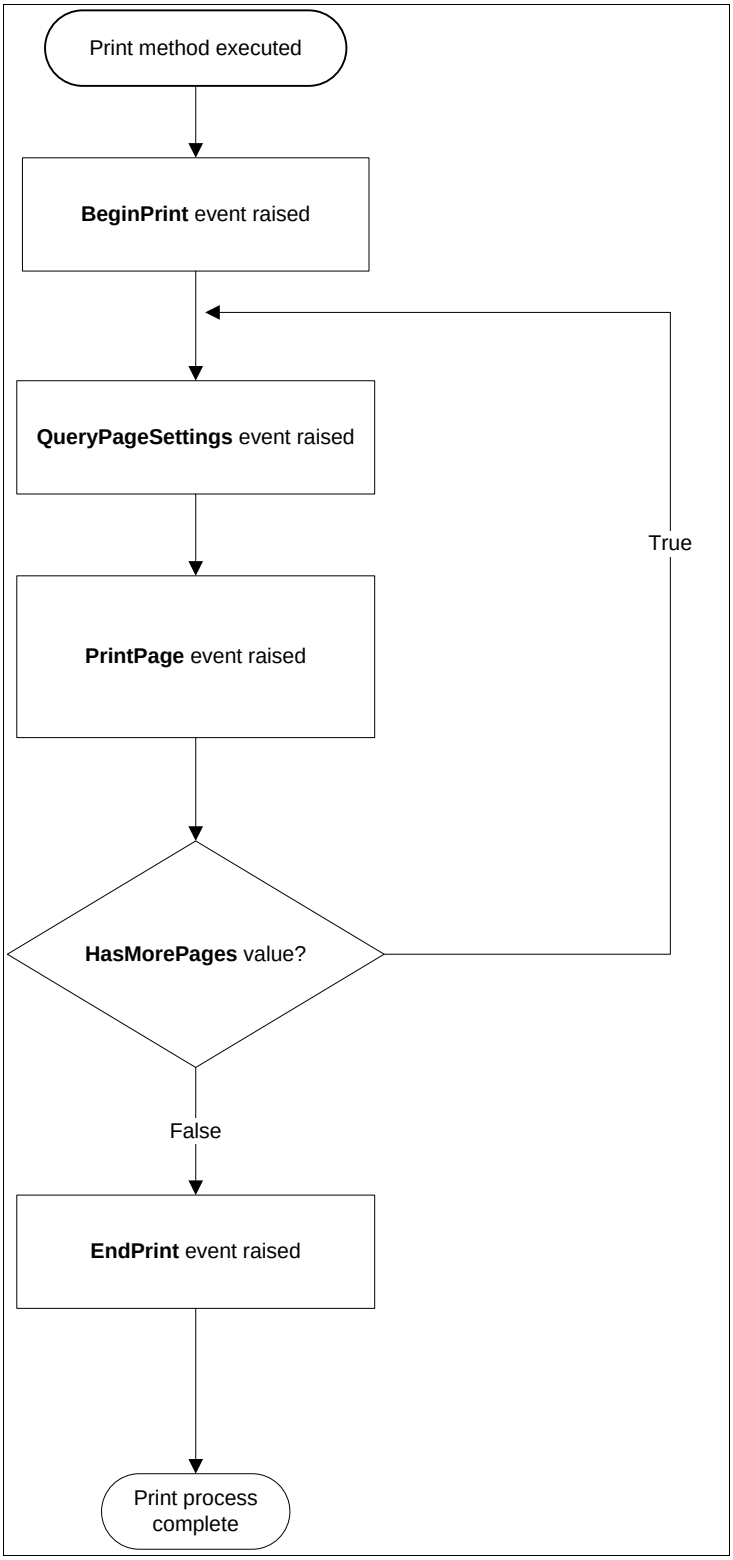


Figure 5 - Overview of the PrintDocument process flow

**The BeginPrint event**

The `BeginPrint` event is triggered once at the point when the print method is executed. The event provides an argument of type `Printing.PrintEventArgs` that provides the following properties:

- `Cancel` – Set this property to True if you wish to cancel the print operation.
- `PrintAction` – This tells you what kind of print is being performed. There are three possible values:
    - `PrintAction.PrintToFile` – The document is being printed to a file.
    - `PrintAction. PrintToPreview`– The document is being printed to a print preview window.
    - `PrintAction. PrintToPrinter`– The document is being printed to a print device.

It is usual to initialise any variables that are going to be used in tracking the print process in the `BeginPrint` event. The following sets an integer to track the current page being printed.

```
Private Sub PrintDocument_BeginPrint(ByVal sender As Object, _
          ByVal e As System.Drawing.Printing.PrintEventArgs) _
                   Handles PrintDocument.BeginPrint

    'Initialise the global settings in this event handler
    _currentPageIndex = 0

End Sub
```

#### The QueryPageSettings event

The `QueryPageSettings` event is raised once before each page is printed. The event provides an argument of type `QueryPageSettingsEventArgs` that allows you to check and, if required, modify the `PageSettings` that will be applied to the next page to be printed.

The properties of this argument that can be changed to alter the way the next page is printed are:

- `Color` - Set this to False to force the page to be printed in black and white, or True to print the next page in colour.

- `Landscape` - Set this to False to print the next page in portrait mode (with the longest edge of the paper in the vertical plane) or True to print the next page in landscape mode (with the longest edge of the paper in the horizontal plane).

- `PaperSize` – Allows you to get or change the paper size used to print the next page. If you are changing this value you should check that the new value is a paper size that the print device supports or a run time error will occur.

- `PaperSource` – Allows you to get or change tray from which the paper for the next page is to be taken. If you are changing this value you need to check the paper sources that the print device supports or a run time error will occur. The possible paper sources are:

    - `AutomaticFeed` - The paper is fed in from an automatic feed tray.

    - `Cassette` - The paper is fed from a paper cassette loader.

    - `Envelope` - The next page is printed on an envelope fed from the automatic envelope loader.

    - `FormSource` - The next page is taken from the printer's default paper bin.

    - `LargeCapacity` - The next page is taken from the printer's high capacity paper bin.

    - `LargeFormat` - The next page is taken from the large size paper bin. For example in the case of a laser printer that has a tray for A4 and a separate tray for A3 paper this would indicate that the next sheet should be taken from the A3 bin.

    - `SmallFormat`- The next page is taken from the smaller size paper bin. For example in the case of a laser printer that has a tray for A4 and a separate tray for A3 paper this would indicate that the next sheet should be taken from the A4 bin.

- `Upper`- The next page should be taken from the topmost paper bin.
- `Middle` - The next page should be taken from the middle paper bin.
- `Lower` - The next page should be taken from the lower paper bin.
- `Manual` – The next page should be taken from the manual feed area. In practice this is most often used for pages to be printed on specific headed paper.
- `ManualFeed` - The next page is printed on an envelope fed from the manual envelope feed area.
- `TractorFeed` - The next page is printed paper from a tractor feed (usually this is a continuous roll of paper with punched holes that are driven by a tractor mechanism on the print device).
- `Custom` – Some printers support additional trays that are not listed in the above list. For example many newer inkjet printers have a photo tray that takes postcard size paper for printing photographs.

- `PrinterResolution` - Allows you to select the resolution (in terms of number of pixels per given area of page) to use to print the next page. The possible settings are:
  - `High` – Highest possible resolution.
  - `Medium` - The middle resolution.
  - `Low` - Low resolution.
  - `Draft` - Very low resolution for draft printing.
  - `Custom` – The printer resolution is set to a custom resolution. When this printer resolution is selected the actual resolution to use in dots per inch are set using the `PrinterResolution.X` and `PrinterResolution.Y` properties.

  The following example shows how you can set alternate pages to be landscape versus portrait and low versus high resolution accordingly:

```
Private Sub PrintDocument_QueryPageSettings(ByVal sender As Object, _
     ByVal e As System.Drawing.Printing.QueryPageSettingsEventArgs) _
           Handles PrintDocument.QueryPageSettings

   If (_currentPageIndex Mod 2) = 0 Then
     e.PageSettings.Landscape = True
     e.PageSettings.PrinterResolution.Kind = PrinterResolutionKind.High
   Else
     e.PageSettings.Landscape = False
     e.PageSettings.PrinterResolution.Kind = PrinterResolutionKind.Draft
   End If

End Sub
```

#### The PrintPage event

The `PrintPage` event is raised once for each page to be printed until the print subsystem is notified that there are no more pages to print. The event provides an argument of type which has the following properties:

- `Cancel` - Set this property to true to cancel the print job altogether.
- `HasMorePages` - Set this to true to indicate that the print job has more pages to print. Once the `PrintPage` event has completed then the loop of `QueryPageSettings` event and `PrintPage` event will repeat so long as this property is set to true. It is important to recall that the same `PrintPage` event handler will be called for each page so it you need to keep track of how much of your data has been printed in order to decide whether or not more pages are required.
- `PageBounds` - This property tells you the size of the entire page area is that can be printed on.
- `MarginBounds` - This property gives the size of the page that can be printed on within the page bounds. This property can be changed by the user when they are setting the page settings for the print job so you should endeavour to keep all your print operations within these bounds.
- `PageSettings` - This property gives you the page settings that have been selected for this print page. This can be changed for each page in the `QueryPageSettings` event handler.
- `Graphics` - This is the canvas onto which all the graphics operations which will result in the printed page are performed. When the `PrintPage` event handler completes, this graphics item is passed back to the printer driver to convert into the graphics instructions that the print device uses to print the page.

#### Printing images

Images are printed using the `Graphics.DrawImage` method. There are 30 overloaded versions of this method, but perhaps the most straight-forward and useful one takes the source image and the target rectangle, and the image is then stretched (or squashed, if the target rectangle is smaller than the source image) to fit:

```
Public Sub DrawImage(image As Image, rect As Rectangle )
```

However, a more usual requirement is to put the image in the target and either centre it in the target or clip it depending on whether it is larger or smaller than the place it is

being put. This can be done by the overloaded version of the DrawImage method that specifies the rectangular part of the source image (in the variable **img**) that is copied into the target rectangle.

To make this code a little bit more readable I have added enumerated types to indicate the desired horizontal and vertical alignments of the printed image:

```
Public Enum VerticalAlignments
    Top = 1
    Centre = 2
    Bottom = 3
End Enum

Public Enum HorizontalAlignments
    Left = 1
    Centre = 2
    Right = 3
End Enum

'If the image is larger than the printable area, clip it,
'otherwise align it within the printable area
Dim xOffset As Integer, yOffset As Integer
Select Case VerticalAlignment
  Case VerticalAlignments.Top
       'Align image at top of printable area
       yOffset = 0
  Case VerticalAlignments.Centre
       'Align image at the middle of printable area
       yOffset = CInt((img.Height / 2) _
                 - (BoundingRectangle.Height / 2))
  Case VerticalAlignments.Bottom
      'Align image at bottom of printable area
      yOffset = img.Height - BoundingRectangle.Height
End Select
Select Case HorizontalAlignment
  Case HorizontalAlignments.Left
       xOffset = 0
  Case HorizontalAlignments.Centre
       xOffset = CInt((img.Width / 2) - _
          (BoundingRectangle.Width / 2))
  Case HorizontalAlignments.Right
       xOffset = img.Width - BoundingRectangle.Width
End Select

Dim SourceArea As New Rectangle(0, 0, _
    BoundingRectangle.Width, BoundingRectangle.Height)

If img.Width > BoundingRectangle.Width Then
     SourceArea.X += xOffset
Else
     SourceArea.X -= xOffset
End If

If img.Height > BoundingRectangle.Height Then
```

```
            SourceArea.Y += yOffset
        Else
            SourceArea.Y -= yOffset
        End If

        Canvas.DrawImage(img, BoundingRectangle, SourceArea, _
            GraphicsUnit.Pixel)
```

The output (if the image is smaller than the target rectangle) would be as follows:

Alternatively if the source image is larger than the target rectangle then the output would be as follows:

#### Printing text

Text is printed using the `Graphics.DrawString` method. As with the DrawImage method, there are a large number of overloaded versions of this method. To provide similar functionality to the image drawing routine above for a text item you would use the following code:

```
Dim _Textlayout As New System.Drawing.StringFormat
Dim foreBrush As Brush

foreBrush = New System.Drawing.SolidBrush(Me.ForeColour)

If HorizontalAlignment = HorizontalAlignments.Left Then
    _Textlayout.Alignment = StringAlignment.Near
ElseIf HorizontalAlignment = HorizontalAlignments.Right Then
    _Textlayout.Alignment = StringAlignment.Far
Else
    _Textlayout.Alignment = StringAlignment.Center
End If

Canvas.DrawString(sText, Me.PrintFont, _
          foreBrush, BoundingRectangle, _Textlayout)
```

This snippet of code allows the text in the string variable called `sText` to be printed with the required horizontal alignment within the boundary rectangle specified in `BoundingRectangle`. This is perfectly adequate for situations where you know that the text will always fit within the rectangle but in the real world you need to decide what to do when the text exceeds the area put aside for it.

#### Printing RTF text

The RichTextBox control existed before .NET and when the control was migrated to the .NET framework, no additional printing functionality was added. This means that in order to print a RichTextBox control's content in a what-you-see-is-what-you-get manner you will need to fall back on the Windows API (Müller).

This is done by sending messages to the window process of the rich text box control containing the text you want to print. These messages cause it to output its content to the printed page.

The following structures are required in order to add the extra information to these messages.

```
<StructLayout(LayoutKind.Sequential)> _
Public Structure STRUCT_RECT
    Public left As Int32
    Public top As Int32
    Public right As Int32
```

```
        Public bottom As Int32
    End Structure

    <StructLayout(LayoutKind.Sequential)> _
    Public Structure STRUCT_CHARRANGE
        Public cpMin As Int32
        Public cpMax As Int32
    End Structure

    <StructLayout(LayoutKind.Sequential)> _
    Public Structure STRUCT_FORMATRANGE
        Public hdc As IntPtr
        Public hdcTarget As IntPtr
        Public rc As STRUCT_RECT
        Public rcPage As STRUCT_RECT
        Public chrg As STRUCT_CHARRANGE
    End Structure

    <StructLayout(LayoutKind.Sequential)> _
    Public Structure STRUCT_CHARFORMAT
        Public cbSize As Integer
        Public dwMask As UInt32
        Public dwEffects As UInt32
        Public yHeight As Int32
        Public yOffset As Int32
        Public crTextColor As Int32
        Public bCharSet As Byte
        Public bPitchAndFamily As Byte
        <MarshalAs(UnmanagedType.ByValArray, SizeConst:=32)> _
        Public szFaceName As Char()
    End Structure
```

You send messages to the rich text box using the SendMessage API call:

```
<DllImport("user32.dll")> _
    Private Shared Function SendMessage(<[In]()> ByVal hWnd As IntPtr, _
                                        <[In]()> _
                                        <MarshalAs(UnmanagedType.U4)> _
                                        ByVal msg As RTFBoxEditMessages, _
                                        <[In]()> ByVal wParam As Int32, _
                                        <[In]()> ByVal lParam As IntPtr) _
                               As Int32

    End Function

    <DllImport("user32.dll")> _
    Private Shared Function SendMessage(<[In]()> ByVal hWnd As IntPtr, _
                                        <[In]()> _
                                        <MarshalAs(UnmanagedType.U4)> _
                                        ByVal msg As RTFBoxEditMessages, _
                                        <[In]()> ByVal wParam As Int32, _
                                        <[In]()> _
                                        <MarshalAs(UnmanagedType.AsAny)> _
                                        ByVal lParam As STRUCT_FORMATRANGE) _
```

```
                                As Int32
    End Function

    <DllImport("user32.dll")> _
    Private Shared Function SendMessage(<[In]()> ByVal hWnd As IntPtr, _
                                <[In]()> _
                                <MarshalAs(UnmanagedType.U4)> _
                                ByVal msg As RTFBoxEditMessages, _
                                <[In]()> ByVal wParam As Int32, _
                                <[In]()> _
                                ByVal lParam As STRUCT_CHARFORMAT) _
                            As Int32

    End Function
```

**The EndPrint event**

The `EndPrint` event is raised after the last `PrintPage` event completes and the `HasMorePages` property has been set to false. This allows you to clean up any variables or resources that you have created for the print job.

Chapter

5

# Creating a .NET print library

Having seen the basic functions used in printing from the .NET framework, and given the probability that you will be using these in a number of different applications, it makes sense to assemble your own print library. In this chapter I will outline a basic framework for a print library which you should be able to adapt to your own requirements.[i]

### High level overview: The components of a structured print document

To create a print library the first step is to outline the main components that will be required to perform printing from the very highest level view. For any data driven document the class hierarchy might look something like this:

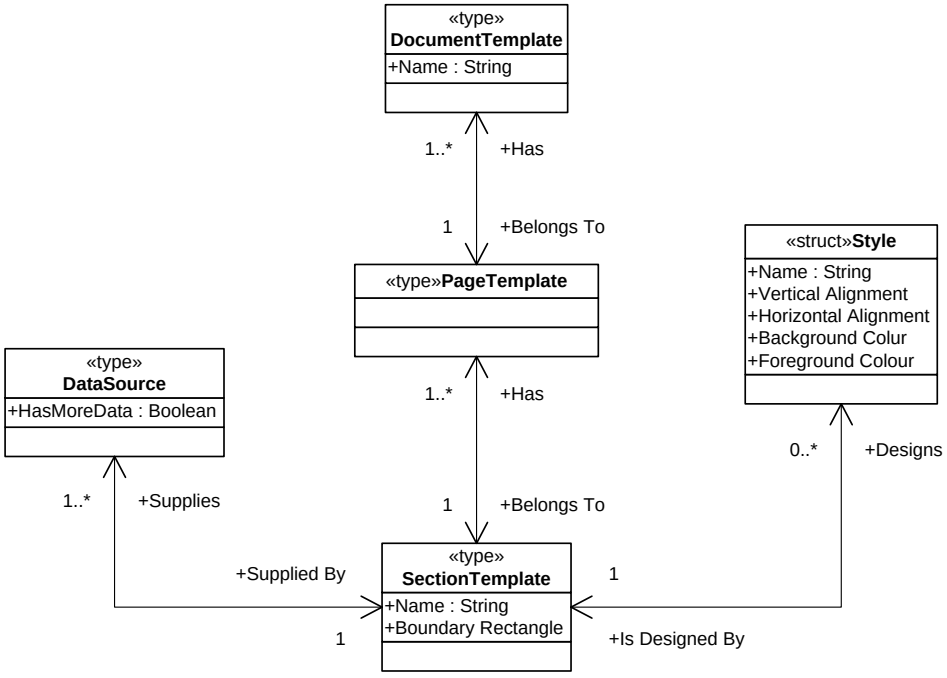


Figure 6 - Print Library Class Hierarchy

This class hierarchy outlines some "common sense" that underpins a print library – a document is made of one or more pages. A page contains one or more sections, which each print from a data source and which are printed using a defined style.

#### The Document Template

The document template is the very top level of the class hierarchy, as what we are interested in printing are documents. It does not require many properties – maybe a name to use to identify the document in the print job queue, but it does have to serve as the interaction layer between the print subsystem and the application. This is done using the `PrintDocument` class (as outlined in Chapter 4).

```
Public Class StructuredPrintDocument

        Private WithEvents _PrintDocument As New PrintDocument


        Private Sub _PrintDocument_BeginPrint(ByVal sender As
Object, ByVal e As System.Drawing.Printing.PrintEventArgs) Handles
_PrintDocument.BeginPrint
            'Initialise the global settings

            'Set all the collection record pointers back to the
first record...

            'Set the document to the first page
        End Sub


        Private Sub _PrintDocument_PrintPage(ByVal sender As Object,
ByVal e As System.Drawing.Printing.PrintPageEventArgs) Handles
_PrintDocument.PrintPage

            'Print the current page

            'If we are not at the end of the document, set
"hasMorePages" to true
        End Sub


        Private Sub _PrintDocument_QueryPageSettings(ByVal sender As
Object, ByVal e As
System.Drawing.Printing.QueryPageSettingsEventArgs) Handles
_PrintDocument.QueryPageSettings
            'Set the page settings according to the current
StructuredPrintPage
        End Sub

        Private Sub _PrintDocument_EndPrint(ByVal sender As Object,
ByVal e As System.Drawing.Printing.PrintEventArgs) Handles
_PrintDocument.EndPrint

        End Sub
 End Class
```

Each document template is made of one or more page templates that dictate the actual layout and content printed on the page.

#### The Page Template

The page template contains all the properties required to define a single *logical* page in the document.

**Logical vs. Physical pages**

A logical page is a template that defines a page. However if the data that you intend to print on that logical page are too big to fit then it is quite possible for one logical page template to produce multiple physical pages in the finished document.

The page template is used in two places in the print process[6]: first in the `QueryPageSettings` event it is used to set up the properties that will be applied to the page to be printed, and then in the `PrintPage` event it performs the actual drawing actions required to print the page itself.

At a minimum this class should provide properties to set the paper size to use, and whether or not to print the page in landscape format.

Each page is further divided into one or more sections (for simplicity these are often defined as rectangular sections on the page) each of which prints a single type of thing – for example a picture, text or a rich text formatted block. These are defined by the section template.

#### The Section Template

Each section of the page is defined by three properties: where it fits on the page, what it is printing and how it is printing the section. In other words: location, data and method.

**Location:**

The bounding rectangle of the section can be an absolute value (defined in the print unit of 1/10ths of a millimetre) or a relative value (defined as a percentage of the page width and height). In the latter case you will need to convert the bounding rectangle to print units before performing the print method.

**Data:**

The data is what is being printed. This can be of any type so long as it is possible to convert the data source into a data type that the print method can handle. For example if your print method requires data as a string you can use the `ToString` method but if it requires an image then you may need to implement custom code to derive an image from the data.

In addition the section template needs to track how much of the data has been printed. This allows for the data to be printed over a number of pages if it overflows the bounding area set for it.

---

[6] See Chapter 4 for a description of the print process

**Method:**

This is the code that draws the data on the page at the given location. In Chapter 4 there are methods to print a picture, text or a rich text formatted area. You could also add methods for printing barcodes, graphs, charts and all sorts of other method as you require them.

Chapter

6

# Getting and setting printer settings in .NET

**Setting printer properties using the System.Printing namespace**

To set printer properties in .NET 3.0, you use the `Printing.PrintQueue` class. (In the framework the properties of the printer and the print queue attached to that printer are both controlled by the same class).

```
Public Sub ChangePrinterComment(ByVal PrinterName As String, _
                                ByVal NewComment As String)

      ' Get access to the local print server
      Dim printServer As New Printing.LocalPrintServer()

      ' Get a reference to the named printer
      Dim pq As New Printing.PrintQueue(printServer, _
                                        PrinterName, _
          Printing.PrintSystemDesiredAccess.AdministratePrinter)


      ' Set the printer comment
      pq.Comment = NewComment

      ' Commit changed properties to the printer subsystem
      pq.Commit()

End Sub
```

The first step is to get an instance of the `PrintServer` class that provides access to the entire print subsystem. The `System.Printing` namespace provides two classes for this: `LocalPrintServer` which is used to access the print system on the local machine, and `PrintServer` which is used to access the print system on a remote print server.

From this you can get an individual `PrintQueue` class which represents the properties and print queue associated with the given named printer. In order to make changes to the printer settings you need to request the `AdministratePrinter` access privilege.

You can then change the printer settings and you then have to execute the `Commit` method to save these changed properties.

If the user credentials under which your code is running do not have sufficient privilege to change the printer properties a trappable error of type

`PrintCommitAttributesException` will occur when you attempt to commit any changes.

#### Setting printer properties using the Windows API

To perform the same change as above in a version of the .NET framework prior to 3.0 means falling back on the windows API – specifically the `OpenPrinter`, `GetPrinter`, `SetPrinter` and `ClosePrinter` API calls.

```
<DllImport("winspool.drv", EntryPoint:="OpenPrinter", _
    SetLastError:=True, CharSet:=CharSet.Unicode, _
    ExactSpelling:=False, _
    CallingConvention:=CallingConvention.StdCall)> _
Public Function OpenPrinter( _
        <InAttribute()> ByVal pPrinterName As String, _
        <OutAttribute()> ByRef phPrinter As IntPtr, _
        <InAttribute()> ByVal pDefault As PRINTER_DEFAULTS_
                                        ) As Boolean

    End Function

<DllImport("winspool.drv", EntryPoint:="ClosePrinter", _
SetLastError:=True, _
ExactSpelling:=True, _
CallingConvention:=CallingConvention.StdCall)> _
Public Function ClosePrinter(<InAttribute()> ByVal hPrinter As IntPtr) As
Boolean

    End Function
```

And the structure passed to is defined as:

```
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Unicode)> _
    Friend Class PRINTER_DEFAULTS
        Public lpDataType As Int32
        Public lpDevMode As Int32
        <MarshalAs(UnmanagedType.U4)> Public DesiredAccess As
PrinterAccessRights

        Public Sub New(ByVal DefaultDesiredAccess As PrinterAccessRights)

            DesiredAccess = DefaultDesiredAccess

        End Sub

    End Class
```

This allows you to select the access rights you want to attempt to get for the printer, from a combination of the following flags.

```
    <FlagsAttribute()> _
    Public Enum PrinterAccessRights
        ' Allowed to read printer information
        READ_CONTROL = &H20000
        ' Allowed to write device access control info
        WRITE_DAC = &H40000
        ' Allowed to change the object owner
```

```
        WRITE_OWNER = &H80000
        ' SERVER_ACCESS_ADMINISTER
        SERVER_ACCESS_ADMINISTER = &H1
        '  SERVER_ACCESS_ENUMERATE
        SERVER_ACCESS_ENUMERATE = &H2
        ' Allows administration of a printer
        PRINTER_ACCESS_ADMINISTER = &H4
        '  Allows printer general use (printing, querying)
        PRINTER_ACCESS_USE = &H8
        ' Allows use and administration.
        PRINTER_ALL_ACCESS = &HF000C
        ' SERVER_ALL_ACCESS = (STANDARD_RIGHTS_REQUIRED |
SERVER_ACCESS_ADMINISTER | SERVER_ACCESS_ENUMERATE)
        SERVER_ALL_ACCESS = &HF0003
    End Enum
```

For the `GetPrinter` and `SetPrinter` API calls there are several different signatures you can use depending on the level of details you want to get or set. This is to allow you to reduce the impact on the system by only asking for the information you are going to require.

The comment property is part of the printer properties can be retrieved/set using the smaller `PRINTER_INFO_1` structure:

```
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Unicode),
System.Security.SuppressUnmanagedCodeSecurity()> _
Friend Class PRINTER_INFO_1
    <MarshalAs(UnmanagedType.U4)> Public Flags As Int32
    <MarshalAs(UnmanagedType.LPWStr)> Public pDescription As String
    <MarshalAs(UnmanagedType.LPWStr)> Public pName As String
    <MarshalAs(UnmanagedType.LPWStr)> Public pComment As String

        Public Sub New(ByVal hPrinter As IntPtr)

            Dim BytesWritten As Int32 = 0
            Dim ptBuf As IntPtr

            ptBuf = Marshal.AllocHGlobal(1)

            If Not GetPrinter(hPrinter, 1, ptBuf, 1, BytesWritten)
Then
                If BytesWritten > 0 Then
                    '\\ Free the buffer allocated
                    Marshal.FreeHGlobal(ptBuf)
                    ptBuf = Marshal.AllocHGlobal(BytesWritten)
                    If GetPrinter(hPrinter, 1, ptBuf, BytesWritten,
BytesWritten) Then
                        Marshal.PtrToStructure(ptBuf, Me)
                    Else
                        Throw New Win32Exception()
                    End If
                    '\\ Free this buffer again
                    Marshal.FreeHGlobal(ptBuf)
```

```
                Else
                    Throw New Win32Exception()
                End If
            Else
                Throw New Win32Exception()
            End If

        End Sub

        Public Sub New()

        End Sub
    End Class
```

However for the full gamut of printer settings you will need to use the much larger PRINTER_INFO_2 structure.

```
    <StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Unicode),
System.Security.SuppressUnmanagedCodeSecurity()> _
Friend Class PRINTER_INFO_2
   <MarshalAs(UnmanagedType.LPWStr)> Public pServerName As String
   <MarshalAs(UnmanagedType.LPWStr)> Public pPrinterName As String
   <MarshalAs(UnmanagedType.LPWStr)> Public pShareName As String
   <MarshalAs(UnmanagedType.LPWStr)> Public pPortName As String
   <MarshalAs(UnmanagedType.LPWStr)> Public pDriverName As String
   <MarshalAs(UnmanagedType.LPWStr)> Public pComment As String
   <MarshalAs(UnmanagedType.LPWStr)> Public pLocation As String
   <MarshalAs(UnmanagedType.U4)> Public lpDeviceMode As Int32
   <MarshalAs(UnmanagedType.LPWStr)> Public pSeperatorFilename _
            As String
   <MarshalAs(UnmanagedType.LPWStr)> Public pPrintProcessor  _
            As String
   <MarshalAs(UnmanagedType.LPWStr)> Public pDataType As String
   <MarshalAs(UnmanagedType.LPWStr)> Public pParameters As String
   <MarshalAs(UnmanagedType.U4)> Public lpSecurityDescriptor _
            As Int32
   Public Attributes As Int32
   Public Priority As Int32
   Public DefaultPriority As Int32
   Public StartTime As Int32
   Public UntilTime As Int32
   Public Status As Int32
   Public JobCount As Int32
   Public AveragePPM As Int32
   Private dmOut As New DEVMODE

        Public Sub New(ByVal hPrinter As IntPtr)

            Dim BytesWritten As Int32 = 0
            Dim ptBuf As IntPtr

            ptBuf = Marshal.AllocHGlobal(1)

            If Not GetPrinter(hPrinter, 2, ptBuf, 1, BytesWritten) Then
```

```
                If BytesWritten > 0 Then
                    '\\ Free the buffer allocated
                    Marshal.FreeHGlobal(ptBuf)
                    ptBuf = Marshal.AllocHGlobal(BytesWritten)
                    If GetPrinter(hPrinter, 2, ptBuf, _
                          BytesWritten, BytesWritten) Then
                        Marshal.PtrToStructure(ptBuf, Me)
                        '\\ Fill any missing members
                        If pServerName Is Nothing Then
                            pServerName = ""
                        End If
                        '\\ If the devicemode is available, get it
                        If lpDeviceMode > 0 Then
                            Dim ptrDevMode As New IntPtr(lpDeviceMode)
                            Marshal.PtrToStructure(ptrDevMode, dmOut)
                        End If
                    Else
                        Throw New Win32Exception()
                    End If
                    '\\ Free this buffer again
                    Marshal.FreeHGlobal(ptBuf)
                Else
                    Throw New Win32Exception()
                End If
            Else
                Throw New Win32Exception()
            End If
        End Sub

        Public ReadOnly Property DeviceMode() As DEVMODE
            Get
                Return dmOut
            End Get
        End Property


    End Class
```

The definition of `GetPrinter` and `SetPrinter` can be overloaded for each structure.

```
<DllImport("winspool.drv", EntryPoint:="GetPrinter", _
     SetLastError:=True, CharSet:=CharSet.Unicode, _
     ExactSpelling:=False, _
     CallingConvention:=CallingConvention.StdCall)> _
Public Function GetPrinter _
            (<InAttribute()> ByVal hPrinter As IntPtr, _
             <InAttribute()> ByVal Level As Int32, _
             <OutAttribute()> ByVal lpPrinter As IntPtr, _
             <InAttribute()> ByVal cbBuf As Int32, _
             <OutAttribute()> ByRef lpbSizeNeeded As Int32) As Boolean

    End Function
```

```
<DllImport("winspool.drv", EntryPoint:="SetPrinter", _
SetLastError:=True, CharSet:=CharSet.Unicode, _
ExactSpelling:=False, _
CallingConvention:=CallingConvention.StdCall)> _
Public Function SetPrinter _
   (<InAttribute()> ByVal hPrinter As IntPtr, _
  <InAttribute(), MarshalAs(UnmanagedType.U4)> ByVal Level As
PrinterInfoLevels, _
  <InAttribute(), MarshalAs(UnmanagedType.LPStruct)> ByVal pPrinter As
PRINTER_INFO_1, _
     <InAttribute(), MarshalAs(UnmanagedType.U4)> ByVal Command As
PrinterControlCommands) As Boolean

    End Function
```

When calling `SetPrinter` you need to specify what type of command you are setting

```
    Public Enum PrinterControlCommands
        ' - Update the printer info data
        PRINTER_CONTROL_SETPRINTERINFO = 0
        ' Pause the printing of the currently active job
        PRINTER_CONTROL_PAUSE = 1
        ' Resume printing if paused
        PRINTER_CONTROL_RESUME = 2
        ' Terminate and delete the currently printing job
        PRINTER_CONTROL_PURGE = 3
        ' Set the printer status for the current job
        PRINTER_CONTROL_SET_STATUS = 4
    End Enum
```

These API calls are then used as follows:

```
Public Sub ChangePrinterComment(ByVal PrinterName As String, _
                                ByVal NewComment As String)

   Dim pd As New PRINTER_DEFAULTS(PrinterAccessRights.PRINTER_ALL_ACCESS)
   Dim phPrinter As IntPtr

   If OpenPrinter(PrinterName, phPrinter, pd) Then
            Dim printerInfo As New PRINTER_INFO_1(phPrinter)

            printerInfo.pComment = NewComment

            SetPrinter(phPrinter, PrinterInfoLevels.PrinterInfoLevel1, _
                    printerInfo, _
                    PrinterControlCommands.PRINTER_CONTROL_SETPRINTERINFO)

            'Close the printer handle once it is used
            ClosePrinter(phPrinter)
   End If

 End Sub
```

#### Listing the print jobs on a printer using the System.Printing namespace

Each printer on a print server is linked to an instance of a `PrintQueue` class. You can list all the printers on a server by using the `PrintServer.GetPrintQueues` method:

```
Private Sub RefreshPrintersList()

     Me.ComboBox_Printers.Items.Clear()

     Dim thisServer As System.Printing.PrintServer = New PrintServer( _
       PrintSystemDesiredAccess.EnumerateServer)

     For Each thisPrintQueue As PrintQueue In thisServer.GetPrintQueues()
                Me.ComboBox_Printers.Items.Add(thisPrintQueue)
     Next

End Sub
```

The `PrintQueue` class has `GetPrintJobInfoCollection` a method which is used to list the jobs currently on that print queue. However it is important to note that the `PrintQueue` class does not represent a live view of the print queue and must be updated by executing the `Refresh` method before getting the job list:

```
Public Sub RefreshPrintJobList(ByVal thisPrintQueue As PrintQueue)

  Me.ListBox_PrintJobs.Items.Clear()

  If thisPrintQueue.NumberOfJobs > 0 Then
      'Update the print queue from the spooler

   thisPrintQueue.Refresh()

    Dim allJobs As PrintJobInfoCollection =
thisPrintQueue.GetPrintJobInfoCollection()
    For Each thisJob As PrintSystemJobInfo In allJobs
         Me.ListBox_PrintJobs.Items.Add(thisJob)
    Next
  End If

End Sub
```

#### Listing the print jobs on a printer using the Windows API

To list the print jobs on a printer in a .NET framework version prior to 3.0 requires you to fall back on the Windows API. The API calls for `OpenPrinter` and `ClosePrinter` are as listed above, and in addition to these you need to use

```
    <DllImport("winspool.drv", EntryPoint:="EnumJobs", _
 SetLastError:=True, CharSet:=CharSet.Unicode, _
 ExactSpelling:=False, _
 CallingConvention:=CallingConvention.StdCall)> _
Public Function EnumJobs _
         (<InAttribute()> ByVal hPrinter As IntPtr, _
          <InAttribute()> ByVal FirstJob As Int32, _
```

```
            <InAttribute()> ByVal NumberOfJobs As Int32, _
 <InAttribute(), MarshalAs(UnmanagedType.U4)> ByVal Level As
JobInfoLevels, _
             <OutAttribute()> ByVal pbOut As IntPtr, _
             <InAttribute()> ByVal cbIn As Int32, _
             <OutAttribute()> ByRef pcbNeeded As Int32, _
              <OutAttribute()> ByRef pcReturned As Int32 _
    ) As Boolean


    End Function
```

As is the case with `GetPrinter`, there are a number of different structures you can ask the `EnumJobs` function to return depending on the level of information you need about them.

```
    ' The level (JOB_LEVEL_n) structure to read from the spooler
    Public Enum JobInfoLevels
        ' Read a JOB_INFO_1 structure
        JobInfoLevel1 = &H1
        ' Read a JOB_INFO_2 structure
        JobInfoLevel2 = &H2
        ' Read a JOB_INFO_3 structure
        JobInfoLevel3 = &H3
    End Enum
```

For this example we will get a small subset of the possible print job information by using `JOB_INFO_1`

```
   <StructLayout(LayoutKind.Sequential)> Friend Class SYSTEMTIME
        Public wYear As Int16
        Public wMonth As Int16
        Public wDayOfWeek As Int16
        Public wDay As Int16
        Public wHour As Int16
        Public wMinute As Int16
        Public wSecond As Int16
        Public wMilliseconds As Int16

        Public Function ToDateTime() As DateTime

            Return New DateTime(wYear, wMonth, wDay, wHour, wMinute,
wSecond, wMilliseconds)

        End Function

    End Class

    <StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Unicode),
System.Security.SuppressUnmanagedCodeSecurity()> _
    Friend Class JOB_INFO_1
        Public JobId As Int32
```

```
        <MarshalAs(UnmanagedType.LPWStr)> Public pPrinterName As String
        <MarshalAs(UnmanagedType.LPWStr)> Public pMachineName As String
        <MarshalAs(UnmanagedType.LPWStr)> Public pUserName As String
        <MarshalAs(UnmanagedType.LPWStr)> Public pDocument As String
        <MarshalAs(UnmanagedType.LPWStr)> Public pDatatype As String
        <MarshalAs(UnmanagedType.LPWStr)> Public pStatus As String
        <MarshalAs(UnmanagedType.U4)> Public Status As Int32
        Public Priority As Int32
        Public Position As Int32
        Public TotalPage As Int32
        Public PagesPrinted As Int32
        <MarshalAs(UnmanagedType.Struct)> Public Submitted As SYSTEMTIME

        Public Sub New()
            '\\ If this structure is not "Live" then a printer handle and
job id are not used
        End Sub


        Public Sub New(ByVal lpJob As IntPtr)
            Marshal.PtrToStructure(lpJob, Me)
        End Sub

    End Class
```

The resulting code to get the list of print jobs on the named printer is therefore:

```
Public Function RefreshPrintJobList(ByVal PrinterName As String) As
List(Of JOB_INFO_1)

        Dim ret As New List(Of JOB_INFO_1)

        Dim pd As New
PRINTER_DEFAULTS(PrinterAccessRights.PRINTER_ACCESS_USE)
        Dim phPrinter As IntPtr

        If OpenPrinter(PrinterName, phPrinter, pd) Then
            Dim pcbNeeded As Int32 '\\ Holds the requires size of the
output buffer (in bytes)
            Dim pcReturned As Int32 '\\ Holds the returned size of the
output buffer (in bytes)
            Dim pJobInfo As IntPtr

            Dim jobCount As Integer = 255

            If Not EnumJobs(phPrinter, 0, jobCount, _
                            JobInfoLevels.JobInfoLevel1, _
                        New IntPtr(0), 0, pcbNeeded, pcReturned) Then
                If pcbNeeded > 0 Then
                    pJobInfo = Marshal.AllocHGlobal(pcbNeeded)
                    Dim pcbProvided As Int32 = pcbNeeded
                    Dim pcbTotalNeeded As Int32 '\\ Holds the requires
size of the output buffer (in bytes)
                    Dim pcTotalReturned As Int32 '\\ Holds the returned
size of the output buffer (in bytes)
```

```
                    If EnumJobs(phPrinter, 0, jobCount, _
                           JobInfoLevels.JobInfoLevel1, pJobInfo, _
                      pcbProvided, pcbTotalNeeded, pcTotalReturned) Then
                        If pcTotalReturned > 0 Then
                            Dim item As Int32
                            Dim pnextJob As IntPtr = pJobInfo
                            For item = 0 To pcTotalReturned - 1
                                Dim jiTemp As New JOB_INFO_1(pnextJob)
                                ret.Add(jiTemp)
                                pnextJob = New IntPtr(pnextJob.ToInt32 +
64)
                            Next
                        End If
                    Else
                        Throw New Win32Exception()
                    End If
                    Marshal.FreeHGlobal(pJobInfo)
                End If
            End If

            'Close the printer handle once it is used
            ClosePrinter(phPrinter)
        End If

        Return ret

    End Function
```

These examples demonstrate that the new System.Printing namespace hides a lot of the complexity of dealing with the print spooler but that it is possible to fall back on the Windows API if you are not able to use the .NET framework version 3.0 or higher.

# Index

# Figures



# Reference

Müller, M. (n.d.). *Getting WYSIWYG Print Results from a .NET RichTextBox*. Retrieved from MSDN: http://msdn2.microsoft.com/en-us/library/ms996492.aspx

[i] The full source code for a VB.Net class library that implements this can be found at: http://www.codeproject.com/KB/printing/printdocumentutilities.aspx