

Creating PDF documents with Synergy

Introduction

This tutorial will show you how easy it is to create PDF documents from Synergy applications. The tutorial uses the Open Source HARU PDF API which is available for download at <http://libharu.org>, you do not need to download the API as a version of this API has been created for Synergy Language and is available in the Synergy CodeExchange. That version is included with the tutorial.

This tutorial assumes that you have an existing application that generates documents that are printed on pre-printed forms. This tutorial provides a sample text file that represents an invoice that you may have generated in your application.

The tutorial will take you through the process of creating a PDF from an existing text file, and then adding graphical elements to the PDF document.

To complete this tutorial your system must have the following components installed:

- Synergy/DE version 9.5.3b or greater installed.
- Haru PDF

Tutorial Setup

You have been provided with some files that you will use during this tutorial. You will find those files in your **Documents\Synergy Tutorials\CreatingPDF** folder.

Creating PDF documents with Synergy

Creating PDF documents

The task for this tutorial is to create a subroutine that creates a PDF document from an existing text file.

The first task is to locate the provided SynergyDE Workbench workspace file.

- ☐ From the **Windows Start** menu, locate and run the **All Programs→SynergyDE→Workbench**.
- ☐ Once in Workbench, from the **Project** menu select the **Open Workspace** entry.
- ☐ Navigate to the **Synergy Tutorials\CreatingPDF** folder.
- ☐ Select the **PDFtutorial.vpw** file.
- ☐ Click the **Open** button to load the workspace.

The tutorial uses a Synergy/DE Workbench Workspace which includes the following files:

- pdfdbl.dbl – this is a collection of synergy routines that is main Haru PDF API. This is the code that you can download from CodeExchange.
- printPDF.dbl – this is a routine that you will write, that will perform the business logic to convert your text file into a PDF. This file currently contains an empty routine.
- printPDF_drawPage.dbl – this is a routine that you will write, that will add the graphical elements to each page of the PDF document. This file currently contains an empty routine.
- PDFtutorial.dbl – this is a simple test harness
- hpdf_error.def – this contains a list of possible error codes, returned by the Haru PDF API. This file is in the CodeExchange download with pdfdbl.dbl.
- dpc2012.png – an image to add to the PDF document.
- myDoc.ddf – this is the test file that you will be using to create a PDF document.

Creating PDF documents with Synergy

Our first task is to write the code that creates a PDF representation of the text file.

- ☐ In the Workbench, in the **Project Explorer**, expand out the **PDFtutorial.vpj** project and under the **Source Files** folder double-click the **printPDF.dbl** file to open it in the editor window.

In the code edit window, you will see some simple logic to read through a text file, character by character. This logic (in its current form) is non-functional. What you will do now is a series of steps to create a multi-page PDF document

- ☐ Locate the “;; local data” comment and add the following line of code.

```
data pdfDoc           ,@HpdfDoc
data pdfFont          ,@HpdfFont
data pdfPage          ,@HpdfPage
data pageSize         ,HPdfPageSizes
data pageOrientation  ,HPdfPageDirection
data fontSize         ,float
```

These are all the variables that we will be using during this task.

- ☐ Locate the comment “;; create new document” and add the code to create a new PDF document

```
;; create new PDF document
pdfDoc = new HPdfDoc()
```

Now we need to add a new page in the PDF document – but we also need to tell the page which font we need to use. Fonts (and images) are stored once in the document, and then used on each page (as needed).

- ☐ Immediately after the above code, add this code

```
;; set font and page defaults
fontSize = 10.0
pdfFont = pdfDoc.GetFont('Courier', ^null)

pageSize = HPdfPageSizes.HPDF_PAGE_SIZE_LETTER
pageOrientation = HPdfPageDirection.HPDF_PAGE_LANDSCAPE
```

In the above code, we’re populating local variables, as we will use these values in multiple places in the code.

- ☐ Now we need to add the logic to add the new page.

```
pdfPage = pdfDoc.AddPage(pageSize, pageOrientation)
pdfPage.StartPageWithFont(pdfFont, fontSize)
```

At this point, we haven’t started to process the text file; in essence, we have just “opened” the PDF document. Now we need to modify code inside the text file processing loop to deal with line and form breaks.

Creating PDF documents with Synergy

- ☐ Locate the comment “;; start new page”, and add the code to add a new page (again)

```
pdfPage = pdfDoc.AddPage(pageSize, pageOrientation)
pdfPage.StartPageWithFont(pdfFont, fontSize)
```

- ☐ Immediately before the “;; start new page” comment, add this code to end the logical text object that is started by the <page>.StartPageWithFont() method.

```
pdfPage.EndText()
```

- ☐ Locate the comment “;; write out print line”, and add the code to write out a print line.

```
pdfPage.Writes((string) (%atrim(tmpLine)))
```

Now we need to perform the final steps to save the PDF document, when we have finished processing the test file.

- ☐ Locate the comment “;; save document”, and add the following code, to save the PDF document.

```
;; save document
pdfPage.EndText()
pdfDoc.SaveToFile(pdfFile)
clear pdfDoc
```

At this point, you should have a routine that reads through a text file, and uses line terminators and form feeds to create a multi-page PDF document that looks like the original test file.

- ☐ To test the build, from the **Build** menu, select the **Build** entry.
- ☐ Run the program, from the **Build** menu, select the **Execute** entry.
- ☐ There should be a file “myPdfDoc.pdf” in the current folder.

Creating PDF documents with Synergy

Discussion

In this part of the tutorial, we have used some of the basic HARU PDF document functions, plus a couple of Synergex written “helper” methods. The helper methods are **<page>.StartPageWithFont()** and **<page>.Writes()**. They contain the code below.

```
public method StartPageWithFont ,Void
    req in font          ,@HPdfFont
    req in fontsize      ,float
    endparams
    record
        pageheight      ,float
    endrecord
proc
    BeginText()
    SetFontAndSize(font, fontsize)
    pageheight = GetHeight()
    SetLineWidth(1.0)
    SetTextRenderingMode(HPdfTextRenderingMode.HPDF_FILL)
    MoveTextPos(0, pageheight-fontsize) ;; move to top of page
    mreturn
endmethod

public method Writes      ,int
    req in text           ,string
    endparams
    record
        font_size        ,float
    endrecord
proc
    ShowText (text)
    font_size = GetCurrentFontSize()
    MoveTextPos(0.0, -font_size)        ;; move down to next row
    mreturn 0
endmethod
```

As you can see, the **StartPageWithFont()** method executes many functions, including the **<page>.BeginText()** method. As we added the logic to save the PDF document, we called the method **<page>.EndText()**, but we didn't type the corresponding **<page>.BeginText()**. As you can see, we actually did perform the **BeginText()**—but as part of the **StartPageWithFont()** method.

One of the issues that you will see as we progress through the tutorial is that the Haru PDF API is very specific on the order that routines must be executed.

Automatically generating PDF file names

The PDF tutorial application is not providing a PDF file name to the printPDF routine, so we are hard coding the name of the PDF document. In the next step we will add code to modify the name (and location) of the PDF file generated.

- ☐ Locate the line of code `pdfFile = 'myPdfDoc.pdf'`. We are going to replace this line with the following code that will use the name of the text file to generate the name of the PDF file.

```
data pNode          ,a256
data pDev           ,a256
data pDir           ,a256
data pName          ,a256
data pExt           ,a256

xcall parse(aPrintFile, 1, pNode, pDev, pDir, pName, pExt)

locase pExt
if(pExt == '.pdf')
    throw new ApplicationException("Filename already has PDF extension")

pdfFile = %atrim(pNode)+%atrim(pDev)+%atrim(pDir)+%atrim(pName)+'.pdf'
```

We have built in a check to make sure that we don't overwrite existing documents with a ".pdf" extension.

- ☐ To test the build, from the **Build** menu, select the **Build** entry.
- ☐ Run the program, from the **Build** menu, select the **Execute** entry.
- ☐ There should be a file "myDoc.pdf" in the current folder.

Adding Graphical Elements

Our next task is to modify the PDF document that is being generated, so that it contains the graphical elements needed to make it look like a document that was printed on pre-printed stationery. As the routine that we modified can be used to take any text document and create a PDF document, we are going to provide some additional information to allow it to associate any text file, with logic that will print the appropriate graphical elements. So we are going to modify the routine to “pre-print” all the graphical elements as we start each new page, and then “overstrike” the graphical elements with the text from the text file.

- ☐ Open the **printPDF.dbl** file, and find the parameters for the printPDF function. Add a new parameter (the third parameter) as an optional “PDF template” name.

```
function PrintPDF          ,i
  req in      aPrintFile  ,a      ;; Print (text) file
  opt inout   aPdfFile    ,a      ;; PDF filename
  opt in      aPDFTemplate,a      ;; PDF Template name
endparams
```

- ☐ Now we are passing the template name, we need to add the logic to add the graphical elements.
- ☐ Locate the local data statements where we defined the `fontSize` variable, and add a new local data variable.

```
data pageNum              ,int
```

We will be using this variable to facilitate the ability to print different graphical elements on page 1 and page2+ of the document.

- ☐ Locate the first place in the code that executes the **<page>.AddPage()** logic that we added in a previous task. We will add some new code between the **AddPage()** method and the **StartPageWithFont()** method.

```
if(^passed(aPDFTemplate) && aPDFTemplate)
begin
  pageNum = 1
  pdfPage.SetFontAndSize(pdfFont, fontSize)
  xcall Haru_DrawPage(pdfDoc, pdfPage, aPDFTemplate, pageNum)
end
```

This code will ensure that the page font is setup correctly. Note, the **StartPageWithFont()** already does this, but either we need to draw all the graphical elements before we start adding the text, or make a lot more modifications to manage text/font/graphics in code. By setting the font first, the **Haru_DrawPage()** routine can use font metrics to position graphical elements based upon a row/column coordinate system.

Creating PDF documents with Synergy

- ☐ Now locate the second place where we execute the **<page>.AddPage()** logic, and add this code.

```
if(^passed(aPDFTemplate) && aPDFTemplate)
begin
    pageNum += 1
    pdfPage.SetFontAndSize(pdfFont, fontSize)
    xcall Haru_DrawPage(pdfDoc, pdfPage, aPDFTemplate, pageNum)
end
```

This code is almost identical; we're just incrementing the page number, instead of setting it to 1.

- ☐ Now we need to modify the **Haru_DrawPage()** subroutine. Double click on the **printPDF_drawPage.dbl** routine in the Workbench Projects tab.

- ☐ Add a new local record, containing the variables that we will be using in this task.

```
record
    myPdfFont          ,@HpdfFont
    myPageHeight       ,float
    myPageWidth        ,float
    myFontSize         ,float
    myFontWidth        ,int
    hPos               ,float
    vPos               ,float
    hLen               ,float
    vLen               ,float
endrecord
```

- ☐ In this case, the myDoc.ddf file represents an example invoice, so we're going to use "invoice" as the PDF template name. Please add this as the text string that is used by the USING statement.

```
using pdfTemplate select
('invoice'),
begin
```

Now we're going to add the graphical elements to make the invoice look like this.

Creating PDF documents with Synergy



Account	ABC1234
Invoice	12345
Date	May 20th 2012

Synergex International Corp.
2330 Gold Meadow Way
Gold River CA 95670

Acme Software Development
11070 White Rock Rd
Rancho Cordova CA 95670

SKU	Description	Quantity	Rate	Due
PSW9	Professional Series Workbench	10	2999.00	29990.00
RUN9	Synergy/DE Runtime	100	279.00	27900.00
WRUN9	Synergy/DE ReportWriter Runtime	10	99.00	990.00
XFPL9	Synergy/DE xfServerPlus Runtime	100	0.00	0.00
Total				58880.00
Sales Tax				0.00
Total				58880.00

In order to do this, we need to perform the following tasks

- Draw logo
- Draw a box around the information on the top right corner
- Fill the title line with a background color of green
- Draw boxes/lines around the invoice line data
- Draw boxes/lines around the invoice totals

In order to make life a little easier, we are going to assume a fixed pitch font, and calculate the position of graphical elements by using row/column positions. As you are asked to draw each element you will be told the row/column coordinates to use.

However, you will need to convert row/column coordinates into pixel coordinates, by using the following calculations;

Horizontal position = (column * fontWidth * fontSize) / 1000

- Where font width is based on the width of a sizing character

Vertical position = PageHeight – (row * fontSize)

- Note, the bottom of the page is pixel-row 0

Some parameters will require a length offset, so you can calculate this by obtaining the two positions and subtracting one from the other.

If you do not convert row/columns into pixel coordinates, the Haru PDF API will throw errors.

Creating PDF documents with Synergy

- ☐ In order to allow the calculation of coordinates from row/columns, we need to get the dimensions of the page and font. Use the following code to populate the previously defined variables. This code should be added immediately after the PROC statement.

```
myPdfFont = aPdfPage.GetCurrentFont()

myFontSize = aPdfPage.GetCurrentFontSize()
myFontWidth = myPdfFont.GetUnicodeWidth(65)
myPageHeight = aPdfPage.GetHeight()
myPageWidth = aPdfPage.GetWidth()
```

We are using the character “A” (ASCII 65) as a sizing character.

Unlike previous instructions, the following instructions are going to tell you what to do, but will not provide specific code to type. Please refer to the HARU documentation in the folder `.\CreatingPDF\CodeExchange\Doc\Doc\` provided with the tutorial, or online at <http://libharu.sourceforge.net/documentation.html>. Further examples of how to draw lines, draw rectangles, fill rectangles with color, draw images and color text are in an appendix at the end of this tutorial.

Now, inside the begin/end block for the ‘invoice’ template, you will perform the following tasks:

- ☐ Draw an image from row 2 column 6 to row 9 column 30, using the file `dpc2012.png`
 - Don’t forget that the position of the image, is the position of the lower left corner.
- ☐ Draw a rectangle, from row 4, column 101 to row 8 column 130, with a width of 1.0
- ☐ Draw and fill a rectangle from row 15 column 6 to row 17 column 130. The fill color should match the color on the logo image.
- ☐ Draw a rectangle from row 17 column 6 to row 52 column 130
- ☐ Draw a line from row 15 column 23 to row 52 column 23
- ☐ Draw a line from row 15 column 70 to row 52 column 70
- ☐ Draw a line from row 15 column 87 to row 52 column 87
- ☐ Draw a line from row 15 column 112 to row 52 column 112
- ☐ Draw a rectangle from row 52 column 92 to row 58 column 130
- ☐ Draw a line from row 52 column 112 to row 58 column 112
- ☐ Draw a line from row 54 column 92 to row 54 column 130
- ☐ Draw a line from row 56 column 92 to row 56 column 130

Creating PDF documents with Synergy

Now that we've added the graphical elements to the invoice documents, we actually need to tell the PrintPDF routine which template to use (in the main program).

- ☐ Open the **PDFtutorial.dbl** file from the Workbench Projects tab.
- ☐ Locate the call to PrintPDF, and add a third parameter "invoice". Note this parameter needs to match the text that you added in a previous step.

```
sts = PrintPDF('myDoc.ddf', pdfFile, 'invoice')
```

- ☐ To test the build, from the **Build** menu, select the **Build** entry.
- ☐ Run the program, from the **Build** menu, select the **Execute** entry.
- ☐ There should be a new version of the file "myDoc.pdf" in the current folder that has the graphic elements built into the document.

The final step is to add logic to adjust the PDF document for the second and subsequent pages of a document.

- ☐ In the "invoice" section of code of the **printPdf_drawPage.dbl** file, add this code

```
;; continuation page
if (pageNumber > 1)
begin
    data aText          ,a*      , 'Continued...'
    data textWidth      ,float
    data text_font      ,@HpdfFont

    text_font = aPdfDoc.GetFont('Helvetica', ^null)

    ;; display continuation text (row 4, centered & blue)
    aPdfPage.GSave()
    aPdfPage.SetFontAndSize(text_font, 12.0)
    aPdfPage.BeginText()
    textWidth = aPdfPage.TextWidth(aText)
    vPos = myPageHeight - (04 * myFontSize)
    hPos = ((myPageWidth - textWidth) / 2)
    aPdfPage.MoveTextPos(hPos, vPos)
    aPdfPage.SetRGBFill(0.0, 0.0, 0.8)
    aPdfPage.ShowText(aText)
    aPdfPage.EndText()
    aPdfPage.GRestore()
end
```

This code, will add the wording "Continued..." near the top of the invoice and centered on the page. It also changes font name, font size and the font color.

- ☐ To test the build, from the **Build** menu, select the **Build** entry.

Creating PDF documents with Synergy

- ☐ Run the program, from the **Build** menu, select the **Execute** entry.
- ☐ There should be a new version of the file “myDoc.pdf” in the current folder.

Appendix 1

Using Haru PDF

When using the Haru PDF API, it requires the programmer to call API functions in the correct order. Failure to do this, can result in error codes being returned or (in some cases) the API crashing. The error codes are described in `hpdf_error.def`, but documentation on what generates the error codes is non-existent.

The following are code examples of how to perform the tasks described in the tutorial, and execute the Haru PDF API methods in an appropriate order.

Draw Text

```
data aText          ,a*          , 'Continued...'
data hPos           ,float
data vPos           ,float

aPdfPage.BeginText ()
aPdfPage.MoveTextPos (hPos, vPos)
aPdfPage.ShowText (aText)
aPdfPage.EndText ()
```

If you have multiple text string to display, you do not need to repeat the `BeginText` / `EndText` method calls for each text string.

Draw Text, overriding font and color

```
data aText          ,a*          , 'Continued...'
data text_font      ,@HpdfFont
data textWidth      ,float
data hPos           ,float
data vPos           ,float
data cRed ,float
data cGreen ,float
data cBlue ,float

aPdfPage.GSave ()
text_font = aPdfDoc.GetFont ('Helvetica', ^null)
aPdfPage.SetFontAndSize (text_font, 12.0)
aPdfPage.BeginText ()
textWidth = aPdfPage.TextWidth (aText)
aPdfPage.MoveTextPos (hPos, vPos)
aPdfPage.SetRGBFill (cRed, cGreen, cBlue)
aPdfPage.ShowText (aText)
aPdfPage.EndText ()
aPdfPage.GRestore ()
```

In this example, because we are temporarily overriding several graphical settings, the GSave / GRestore methods are used to save and restore the graphical state of the PDF page.

Draw Rectangle

```
data hPos ,float
data vPos ,float
data hLen ,float
data vLen ,float
data lWidth ,float

aPdfPage.SetLineWidth(lWidth)
aPdfPage.Rectangle(hPos, vPos, hLen, vLen)
aPdfPage.Stroke()
```

Draw Rectangle, fill with a color and make the rectangle border the same color

```
data hPos ,float
data vPos ,float
data hLen ,float
data vLen ,float
data lWidth ,float
data cRed ,float
data cGreen ,float
data cBlue ,float

aPdfPage.SetLineWidth(lWidth)
aPdfPage.SetRGBStroke(cRed, cGreen, cBlue)
aPdfPage.SetRGBFill(cRed, cGreen, cBlue)
aPdfPage.Rectangle(hPos, vPos, hLen, vLen)
aPdfPage.FillStroke()
```

Draw Line

```
data hPos ,float
data vPos ,float
data hPos2 ,float
data vPos2 ,float

aPdfPage.MoveTo(hPos, vPos)
aPdfPage.LineTo(hPos2, vPos2)
aPdfPage.Stroke()
```

Draw Image

```
data pdfImage ,@HPdfImage
data hPos ,float
data vPos ,float
data hLen ,float
data vLen ,float
data pngFileName ,string

pdfImage = aPdfDoc.LoadPngImageFromFile(pngFileName)

aPdfPage.DrawImage(pdfImage, hPos, vPos, hLen, vLen)
```

Discussion

When specifying line width, color, font, etc. on a page, you only need to specify the item once, and the setting is remembered for the entire page, or until you set the item value to a new value.

When positioning items on a page that have height & width, the positioning refers to the position of the bottom left corner of the item.

In the above examples

Only the data types are provided, you need to populate the appropriate values.

aPdfPage is defined as a @HPdfPage object.

hPos/vPos variables are absolute positions on the page.

hLen/vLen variables are relative offsets from a previously specified absolute position.

Colors (cRed, cGreen, cBlue) are a number between 0.0 and 1.0, indicating how much of the color should be used. White would have all three set to 1.0, and black would have all three set to 0.0.

Valid width values include 0.5, 1.0 & 2.0