# Table of Contents

Acumen Training

# Combining PostScript Files Into One PDF File

It's a common task on the newsgroups: combining several PostScript files into a single PDF file. How to do it best? Many organizations create a series of PostScript files that then need to be concatenated into a single Acrobat document. (I recently worked with a group that routinely created reports by combining the PostScript from several hundred one-page Word Perfect documents.)

There are several ways to do this, all of them unsatisfactory in one way or another.

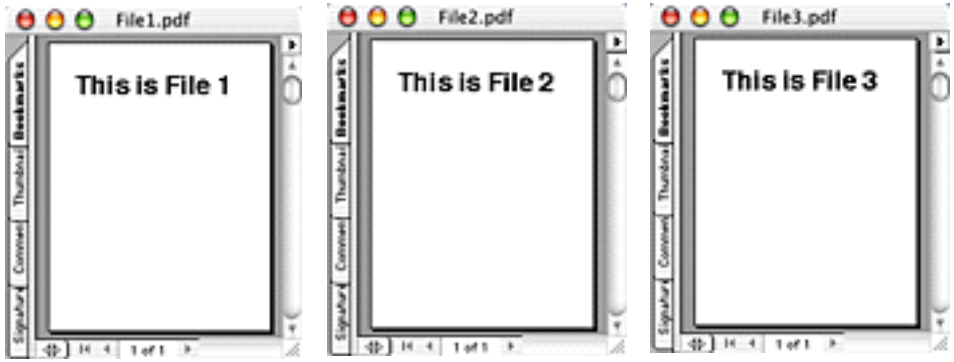
This month, we shall see how to convert several PostScript files into a single PDF file using a little-used Distiller feature called "RunFile."

Next Page ->>

## Concatenating PostScript

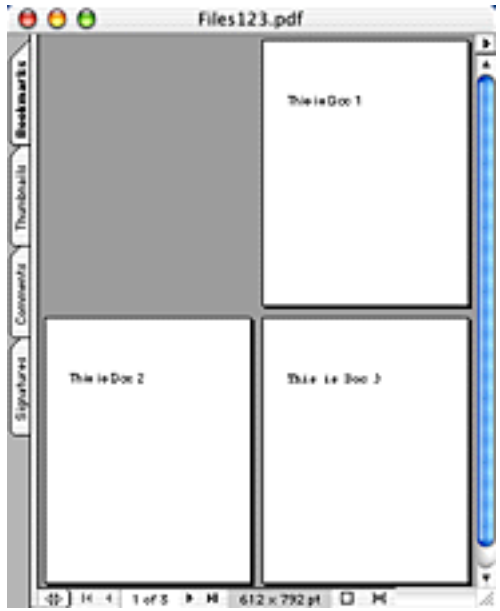
For the purpose of discussion, Let's say you have three PostScript files you need to assemble into a single PDF file. Let's call them *File1.ps, File2.ps,* and *File3.ps.*

If distilled individually, these files produce the one-page output files illustrated at right.




We want to concatenate these three Postscript files so that they become one PDF file, as at right. (If this illustration looks confusing, it's a three-page document viewed in Acrobat's "Continuous Facing Pages" mode.)

Next Page ->

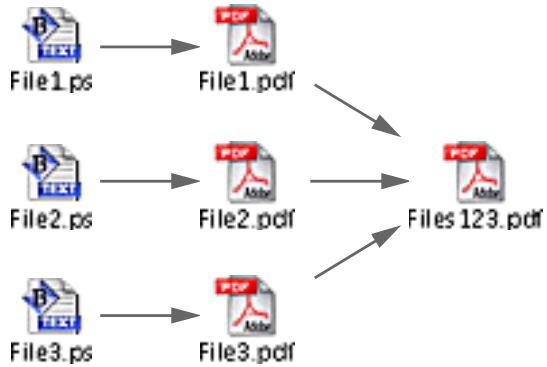


**Concatenating the Files**

There are two approaches we could take to concatenating these files.

*Concatenate the PDF Files*

We could distill the PostScript files separately and then concatenate the resulting PDF files in Acrobat.




All of the files in this article are in a zip file on the Resources page of the Acumen Training website.

We concatenate PDF files in Acrobat by opening the first file in the set and then selecting *Document>Insert Pages,* adding each of the other PDF files, one at a time, to the end of the present file.

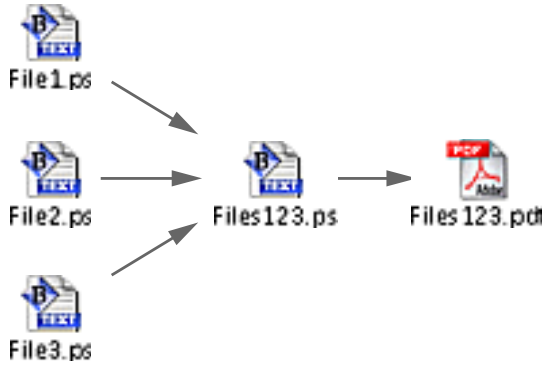
Aside from tedium, there is a serious problem with concatenating the documents this way: if the individual PDF files contain subsetted fonts, the resulting combined PDF file will contain all of the individual subsets, even if they are subsets of the same base font.

In our sample files, *File1.pdf* and *File2.pdf* both contain a subset of Helvetica. The resulting *Files123.pdf* will contain both subsets of Helvetica, even though they were derived from the same font. Carried to extremes, this can greatly expand the size of your concatenated PDF file.

Next Page ->

*Concatenate the PostScript*

The alternative approach is to concatenate the PostScript files and then distill the resulting combined PostScript.




This eliminates the problem with duplicate subset fonts. Distiller will create a single subset for each font based on the characters used in all the PostScript files you are combining. This can result in a dramatically smaller concatenated PDF file.

This is definitely the way you want to make your concatenated PDF file. Unfortunately, combining several multi-megabyte PostScript files into a single file can be a significantly time-consuming activity.

Happily, there is a shortcut, which is the topic of this month's article. (Thought we'd never get to it, didn't you?)

The *Files123.ps* file in the illustration above does not need to contain the complete PostScript taken from the original files. Instead, it needs to contain only a short PostScript program that tells Distiller to distill each of the other files in turn and add them to a single PDF file. I usually think of *Files123.ps* as a *control file,* since it is controlling Distiller, telling it what files to distill.

The name of the "Distill This File" command we shall place in our control file is "RunFile"; *Files123.ps* will have a *RunFile* line for each PostScript file we want to concatenate.

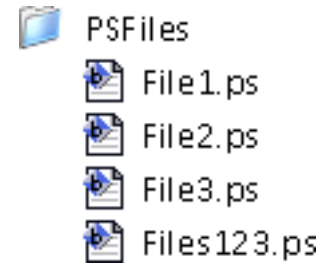
Next Page ->

## Using a Control File

To concatenate a set of PostScript files using a control file, do the following:

1. Place all of the PostScript files you want to concatenate in a single folder on your hard disk.

   Properly speaking, you don't need to do this; it will, however, make life easier for you when you write the control file. You will also ease life's pain a bit if the folder in which you place the PostScript files is not nested inside any other folders. (I place my PostScript files in a folder at the root level of my hard disk, as at right.) You'll see why you want to do this when we discuss writing the control file.




2. Write your control file.

   This is just a text file with a *RunFile* line for each of the files you want to concatenate. We shall look at the exact contents of this file in the next section.

   You can save this file anywhere you wish on your hard disk. However, I usually save it to the folder that contains the PostScript files I'm concatenating.

3. Distill your control program.

   Your control program will direct Distiller to convert all of the PostScript files in your list, adding them to a single PDF file.

Next Page ->

Acumen Journal: Acrobat User 6

**Making a Control File**

A control file is just a text file containing calls to the PostScript *RunFile* operator. You can write it using any text editor. Just open a new file, type in the appropriate lines of PostScript code, and then save the file to disk. You can use a word processor, such as *Microsoft Word,* instead of a text editor, but make sure you save the file as pure text, not as a word processing file.

It's better to write your control file with a text editor, rather than a word processor; it eliminates the chance of occasionally saving your file in the wrong format. For what it's worth, my favorite free or cheap text editors on the Mac and Windows platform are *BBEdit Lite* and *TextPad,* respectively. In Windows, there is also *UltraEdit,* which is quite good. My preferred UNIX editor used to be *emacs,* but since switching to Mac OS X, I find I can use BBEdit for UNIX, too.




Next Page ->

### A *RunFile* Template

Your control file will contain one or two lines of PostScript for each PostScript file you want to concatenate. Here is the control file for concatenating our three sample files:

```
(Wheezy:PSFiles:File1.ps) RunFile
false 0 startjob pop
(Wheezy:PSFiles:File2.ps) RunFile
false 0 startjob pop
(Wheezy:PSFiles:File3.ps) RunFile
```

I suggest you use this file as a template for your own concatenation projects. (A zip file containing all of the files for this article are among the sample files on the Acumen Training website' Resources page: www.acumentraining.com/resources.html).

### *(Disk:Folder:Path) RunFile*

For each PostScript file we want to concatenate, we have a *RunFile* line. The invocation of *RunFile* is preceded, in parentheses, by the full pathname of the PostScript file, starting with the disk name and proceeding through all the folder names to the file. (This is why it's best to not nest your PostScript files too deeply in your hard disk's directory; it is tiresome typing the pathname for a file buried 15 folders deep.)

The exact manner in which you type the pathname is different for the Mac, Windows, and UNIX platforms. We'll come back to this in a moment.

Next Page ->

*false 0 startjob pop*

In between each *RunFile* line, you must place a *startjob* line. This PostScript operator isolates the individual files, keeping them from interfering with each other.

That's all there is to it. Distill the control file and Distiller will combine all of the individual PostScript files into a single PDF file.

**Pathnames**

Each invocation of *RunFile* is preceded by the pathname of a PostScript file. The exact way you express this pathname is different for Macintosh, Windows, and UNIX systems.

On the Mac, pathnames start with the name of the hard drive and folders are separated by colons. Thus, the path to my PostScript files looked something like the following:

```
(Wheezy:PSFiles:File2.ps)
```

In Microsoft Windows, path names start with the drive letter; folders are separated by backslashes. However, because the backslash character has special meaning in PostScript, you must use two backslashes when writing the pathname. (This will be seen by Post-Script as a single backslash.) Thus, a Windows path name would be something like:

```
(C:\\PSFiles\\File2.ps)
```

Finally, on UNIX systems, your pathname would start with a slash (indicating the root directory) and then the subdirectories down to your file, separated by forward slashes:

```
(/PSFiles/File2.ps)
```

Next Page ->

## Conclusion

This technique is pretty easy, once you've done it a few times. It is by far the least troublesome way of combining PostScript files into a single Acrobat file.

If you are an experienced PostScript programmer, you can simplify things a little bit more. However, this is more than I want to go into in this article. Take a look at the file named "RunDirEx.txt" in the *Xtras* folder in your Distiller folder.

Return to Main Menu

# Making Outline Fonts


The most common way of printing outline text in PostScript is to use the *charpath* operator, rather than *show,* to print the text. The *charpath* operator adds the outline of the characters in a string to the current path. For example, the line of PostScript code below has much the same effect as the *show* operator, except it prints the text as an outline.

```
(Binky & Friends) false charpath currentpoint stroke moveto
```

The “currentpoint stroke moveto” paints the outline characters and leaves the currentpoint at the end of the painted text, just as *show* would have done.

There is an alternative approach to printing outline characters; it is relatively easy to create an outline version of any Type 1 font. Printing outline characters can then be done by simply calling *show.*

Let’s see how to do this.

Next Page ->

## PaintType

To create an outline font, we shall take advantage of the *PaintType* entry in a Type 1 font dictionary. *PaintType* is an integer that specifies how a Type 1 font should paint itself. A value of *0* specifies the font should be filled; a value of *2* specifies the font should be stroked.

To convert a Type 1 font to an outline font, all we need to do is change *PaintType* to *2* in the font dictionary. Text in that font will now *show* as outline text.

### *StrokeWidth*

The linewidth that will be used for the stroked character outline is determined by another entry in the font dictionary: *StrokeWidth.* The value associated with this key is the linewidth that should be used for the stroked characters. (If *PaintType* is *0, StrokeWidth* is ignored and may be omitted.)

*StrokeWidth* is expressed in Character Space, so appropriate values will be somewhere around 20 or 30. This width will be scaled down by 1,000 and back up by the point size.

If this sounds vaguely familiar, we mention both *PaintType* and *StrokeWidth* in the PostScript Foundations and PostScript for Support Engineers classes.

Next Page ->

## Making an Outline Font

So, all we need to do to turn a Type 1 font into an outline font is insert a *PaintType* of 2 and an appropriate *StrokeWidth* into the font dictionary. Unfortunately, as you may remember from your PostScript Foundations class, you can't change font dictionaries; they're read-only. What we need to do is create a new font, identical to the original, but with *PaintType* and *StrokeWidth* entries added.

### Changing Fonts

Remembering from our PostScript classes, there are four steps to this process:

1. Create a new dictionary, the same size as the original font, plus room for the additional entries.

2. Copy everything from the original font into the new dictionary. (In PostScript Level 1, you should avoid copying the *FID* entry, but I'm going to ignore this; it's perfectly alright to copy the *FID* in Level 2 and Level 3.)

3. Make the changes you want in your font to the new dictionary. In our case, we'll add our two key-value pairs.

4. Turn the new dictionary into a font dictionary with the *definefont* operator.

Let's see how this applies to making an outline font.

Next Page ->

## Making an Outline Font

Here, we add *PaintType* and *StrokeWidth* to the font ITC Kabel Bold.

All of the files in this article are in a zip file on the Resources page of the Acumen Training website.

```
/ItcKabel-Bold findfont
dup length 1 add dict
begin
currentdict copy
/PaintType 2 def
/StrokeWidth 25 def

/ItcKabel-Bold-O currentdict definefont pop
end

/ItcKabel-Bold-O 60 selectfont

72 600 moveto
(Binky & Friends) show

showpage
```


Let's look at this in detail.

Next Page ->

## The Code in Detail

```
/ItcKabel-Bold findfont
```

Find the font */ItcKabel-Bold.*

```
dup length 1 add dict
begin
```

Create a new dictionary that has a capacity one greater than the ITCKabel font dictionary (reserving space for the new *StrokeWidth* entry). Put that new dictionary on the dictionary stack.

Note that these two lines leave a copy of the original font dictionary on the operand stack (because of the *dup)*.

```
currentdict copy
```

Copy the contents of the original font dictionary (left on the stack) into the current dictionary (which is our newly-created dictionary).

```
/PaintType 2 def
/StrokeWidth 25 def
```

Define the *PaintType* and *StrokeWidth* entries into our new dictionary.

Next Page ->

```
/ItcKabel-Bold-O currentdict definefont pop
end
```

Turn our new dictionary into a font dictionary with the name *ItcKabel-Bold-O.* Then remove our dictionary from the dictionary stack.

```
/ItcKabel-Bold-O 60 selectfont
72 600 moveto
(Binky & Friends) show
```

Use our new font. The output is at right.


Pretty easy, eh?

Next Page ->

**The *findOfont* procedure**

As a final touch, for convenience, let's define a procedure that will act as an alternative to the *findfont* operator, taking a fontname and returning a font dictionary for the outline version of that font. We shall call this procedure *findOfont.*

```
/findOfont     % /OutlineName  /OrigName  =>  <<fdict>>
{
     findfont
     dup length 1 add dict copy
     dup /PaintType 2 put
     dup /StrokeWidth 25 put
     definefont
} bind def

/ItcKabel-Bold-O /ItcKabel-Bold findOfont
60 scalefont setfont

0 0 moveto
(Binky & Friends!) show
```

The *findOfont* procedure takes two names from the operand stack: on top, the name of the original font; beneath, the name of its outline version. It carries out exactly the same steps as were in the previous PostScript example.

Next Page ->

## Composite fonts and outlines

This technique of making outline fonts really shines when you combine the outline font with the original in a composite font. This allows you to switch between the outline and normal styles by placing control codes in the *show* string. (You may remember we discuss using composite fonts for this purpose in the Advanced PostScript class.)

The following code is presented without explanation for now. There will be a Journal article in the next month or two on using Composite fonts for this purpose.

```
/Kabel-Bold                % This will be the name of our composite font
<<   /FontType 0           % Type 0 is a composite font
     /FontMatrix [ 1 0 0 1 0 0 ]
     /FDepVector     [     % Here are the descendent font dictionaries:
          /ItcKabel-Bold findfont
          /ItcKabel-Bold-O /ItcKabel-Bold findOfont
     ]
     /Encoding [ 0 1 ]     % Font 0 is normal; font 1 is outline
     /FMapType 3
>> definefont pop

/Kabel-Bold 24 selectfont

72 600 moveto
% The escape characters in the string switch between fonts 0 and 1.
(We often use \377\001outline\377\000 text.) show
```

We often use outline text.

Next Page ->

The most significant line of PostScript in the preceding program is:

```
(We often use \377\001outline\377\000 text.) show
```

Note that we switch between the regular and outlined characters simply by placing appropriate control characters in the *show* string. This is *way* more efficient than any other method for switching frequently among different fonts; yet, almost no one uses this method.

If you have taken the Advanced PostScript class, you may remember that I tend to rant about how powerful and underused the composite font mechanism is. This is a nice example of that power, but there is simply no time to go into it just now.

Next month, perhaps?

Next Page ->

Acumen Journal: PostScript Tech 19

## Mailbag: Another Use for Double-Slashes

Bob Anderson of Seboomook Scripting, one of my favorite PostScript people, writes with regard to the February 2002 article on the double-slash in PostScript. He gives an excellent additional use of double-slash:

John,

Here is another reason for using // on procedures and other composite objects:

When writing PostScript that modifies the actions of another PostScript file there is always the danger of name conflict. Your name can unintentionally override the same name definition in the [original] PostScript file and visa-versa.

The method I use to eliminate this possibility is to create a private dictionary on the stack and put all my private names and objects in that dictionary. At the end of my procset, I remove the private dictionary from the stack.

The // operator enables my code to access those private objects without performing a name lookup or having the dictionary on the stack.

Next Page ->

```
e.g.
     //proc exec
     //dictionary /name get
     //array # get
     //string show
```

This is especially important when modifying PS code from an unknown source.

Bob Anderson
Seboomook Scripting
bobanderson@mac.com

Next Page ->

Adding a further example to Bob's letter, here we redefine *show* so it spreads out the character spacing in the printed string. We can append PostScript from some other source (*QuarkXpress,* say) to this redefinition and the text in that other PostScript will print with expanded character spacing. (Why would you want to do this? I don't know; it's a simple example.)

```
%%BeginResource: procset PrivateStuff 1 1
/PrivateDict 20 dict def
PrivateDict begin
/dx 1 def
end
%%EndResource

%%BeginResource: procset PublicStuff 1 1
/PublicDict 20 dict def
PrivateDict begin        % so we can refer to our private things
PublicDict begin         % so we can define our public things

/show { //dx 0 ashow } bind def    % dx is found in PrivateDict and its
                                   % value is placed in the show proc.
end             % Remove PublicDict
end             % Remove PrivateDict
%%EndResource

PublicDict begin
...
... Original PostScript from QuarkXpress or other source
...
```

Next Page ->

In this example, our show redefinition uses a constant, dx, that is defined in a private dictionary; this dictionary is not present on the Dictionary stack when the PostScript code we are modifying is executed. Since the dictionary containing dx does not reside on the Dictionary stack, our *dx* definition will not interfere with any dx definitions in the QuarkXpress code whose behavior we are modifying.

Note that the PrivateDict *does* need to be on the dictionary stack when we are defining our *show* procedure so that the double-slash can find the *dx* definition.

A very clever use of double-slash. Thanks, Bob!

Return to Main Menu

# Schedule of Classes, April – July, 2002

Following are the dates and locations of Acumen Training's upcoming PostScript and Acrobat classes. Clicking on a class name below will take you to the description of that class on the Acumen training website.

The PostScript classes are taught in Orange County, California and on corporate sites world-wide. See the Acumen Training web site for more information.

## PostScript Classes

| | | |
|---|---|---|
| **PostScript Foundations** | May 13 – 17 | |
| **Advanced PostScript** | April 29 – May 3 | June 24 – 28 |
| **PostScript for Support Engineers** | April 15 – 19 | June 3 – 7 |
| **Jaws Development** | July 23 – 26 | |

For more classes, go to www.acumentraining.com/schedule.html

**PostScript Course Fees**

PostScript classes cost $2,000 per student.
These classes may also be taught on your organization's site.
Go to www.acumentraining.com/onsite.html for more information.

Registration →
Acrobat Classes →

# Acrobat Class Schedule

**On-Site Only** These classes are taught only on corporate sites. If you have an interest in any of these classes for your group, please see the Acumen Training website regarding arranging an on-site class.

**Acrobat Essentials** This class teaches the student how to make perfect PDF files. It includes complete coverage of the meaning and proper settings of all of the Distiller Job Options.

**Interactive Acrobat** Here we show you how to add bookmarks, links, buttons, sounds, movies, form fields, and other interactive features to an Acrobat file.

**Creating Acrobat Forms** This class shows you how to make interactive forms in Adobe Acrobat. It steps you through creating the form, posting form contents to a server, and everything else you need to create a working PDF form.

**Troubleshooting with Enfocus' PitStop** This class shows the student how to use all of the capabilities of this popular editing and preflight software.

Back to PostScript Classes

Return to First Page

# Contacting John Deubert at Acumen Training

**For more information**

For class descriptions, on-site arrangements or any other information about Acumen's classes:

**Web site:** http://www.acumentraining.com **email:** john@acumentraining.com

**telephone:** 949-248-1241

**mail:** 25142 Danalaurel, Dana Point, CA 92629

**Registering for Classes**

To register for an Acumen Training class, contact John any of the following ways:

**Register On-line:** http://www.acumentraining.com/registration.html

**email:** registration@acumentraining.com

**telephone:** 949-248-1241

**mail:** 25142 Danalaurel, Dana Point, CA 92629

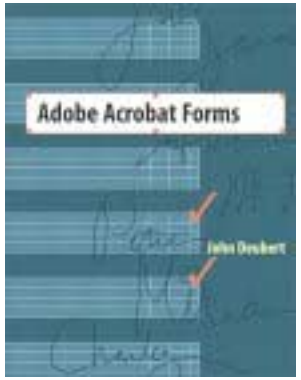
**Back issues**

Back issues of the Acumen Journal are available at the Acumen Training website: www.acumenjournal.com/AcumenJournal.html

Return to First Page

# What's New at Acumen Training?

## Adobe Acrobat Forms




I have just finished writing a book for Peachpit Press on creating Acrobat forms. This book teaches the reader the details of creating forms in Adobe Acrobat. It covers the properties and behaviors of all the Acrobat form fields, the fundamentals of submitting data to a remote program for processing, and everything else you need to collect data from within an Acrobat document.

The Chapter titles are:

1. Introduction
2. Creating Forms: an Overview
3. Basic Interaction: Links
4. The Form Tool
5. Appearances and Actions
6. Buttons
7. Text fields
8. Checkboxes
9. Lists and Combo boxes
10. Radio Buttons
11. Signatures
12. Acrobat Design Tools
13. Rollover Help and Other Tricks
14. Submitting Data
15. The Web, Paper Forms, and Other Topics

A. Creating PDF Files for Acrobat Forms

B. Useful JavaScripts

This book is scheduled to be available On May 1, although you can already pre-order it on Amazon.com.

Return to First Page

# Journal Feedback

If you have any comments regarding the *Acumen Journal,* please let me know. In particular, I am looking for three types of information:

**Comments on usefulness.** Does the Journal provide you with worthwhile information? Was it well written and understandable? Do you like it, hate it? Did it make you regret having ever learned to read?

**Suggestions for articles.** Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

**Questions and Answers.** Do you have any questions about Acrobat, PDF or PostScript? Feel free to email me about. I'll answer your question if I can. If enough people ask the same question, I can turn it into a Journal article.

Please send any comments, questions, or problems to:

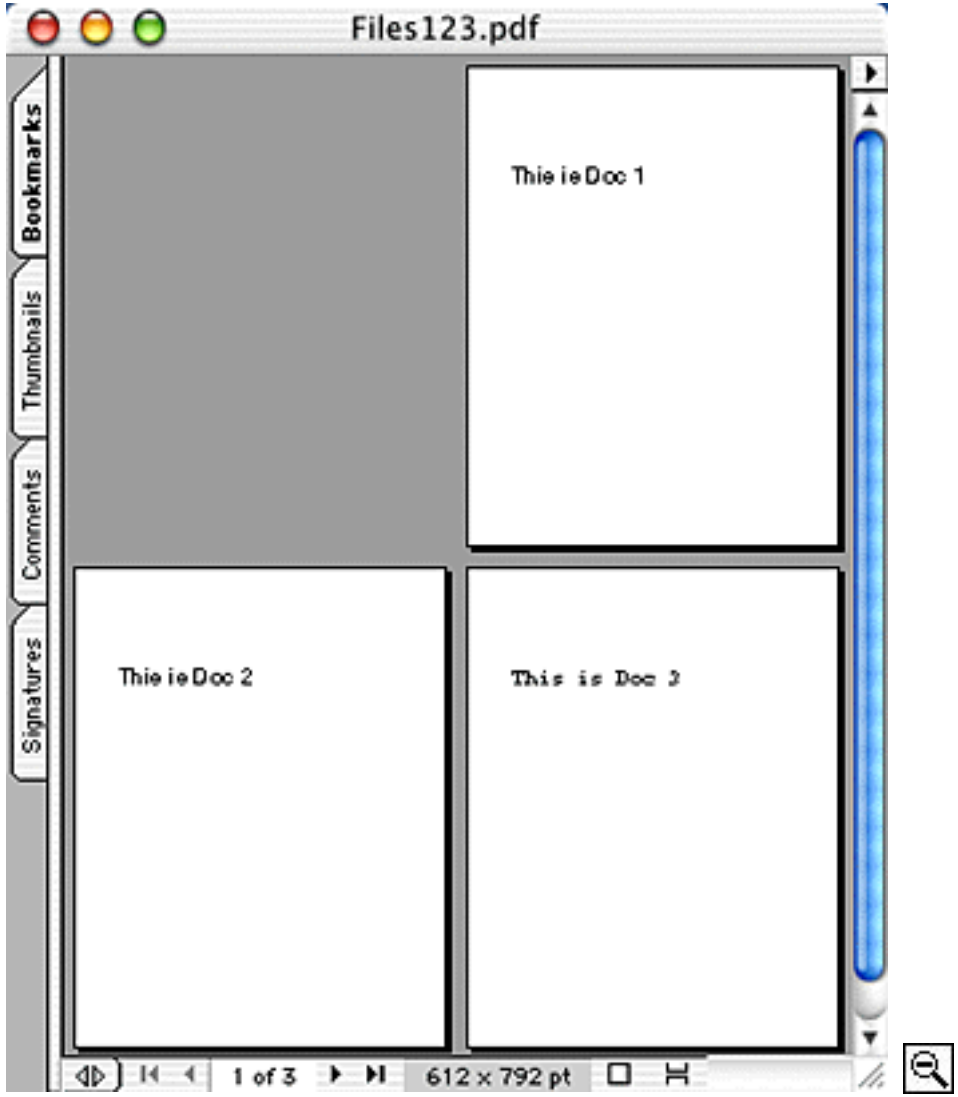
journal@acumentraining.com

Return to Menu


File1.pdf
This is File 1
Bookmarks
Thumbnai
Commen
Signature
1 of 1


File2.pdf
This is File 2
Bookmarks
Thumbnai
Commen
Signature
1 of 1


File3.pdf
This is File 3
Bookmarks
Thumbnai
Commen
Signature
1 of 1


Files123.pdf
Bookmarks
Thumbnails
Comments
Signatures
Thie ie Doc 1
Thie ie Doc 2
This is Doc 3
1 of 3
612 x 792 pt