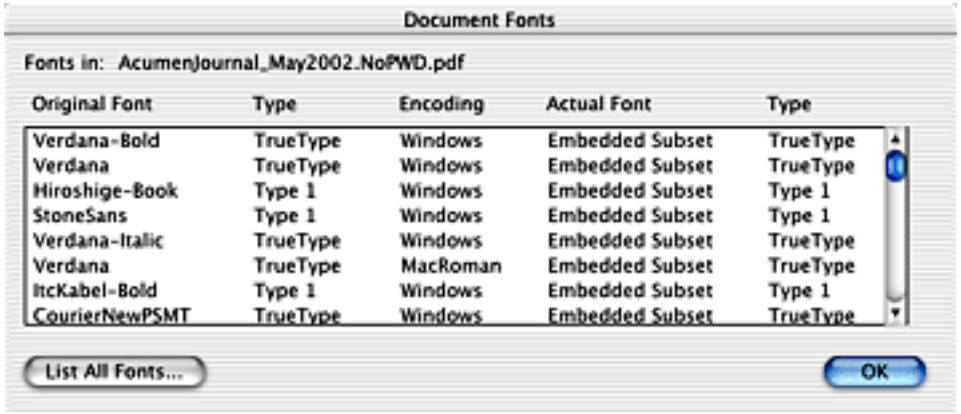
Table of Contents

Acumen Training

Using *Document Fonts* to Avoid Font Problems



PDF files should be self-contained documents, holding everything they need to produce the document they embody. In particular, they should contain the definitions of all of the fonts the document uses. If the fonts are not embedded in the PDF file, a variety of unpleasant things can happen when someone views the file.

Unfortunately, there is no way to determine, by eye, whether or not a PDF file has its fonts embedded. Acrobat goes out of its way to make sure the file looks reasonably good even if the fonts are missing. So how can you tell if you have a file *sans* fonts?

Acrobat provides a *Document Fonts* dialog box that can tell you this. The data it provides needs a little interpretation (it doesn't just say "these fonts are missing, alas") and the dialog box has some limitations. Still, it is the only tool in Acrobat that supplies this information, so we should know how to use it.

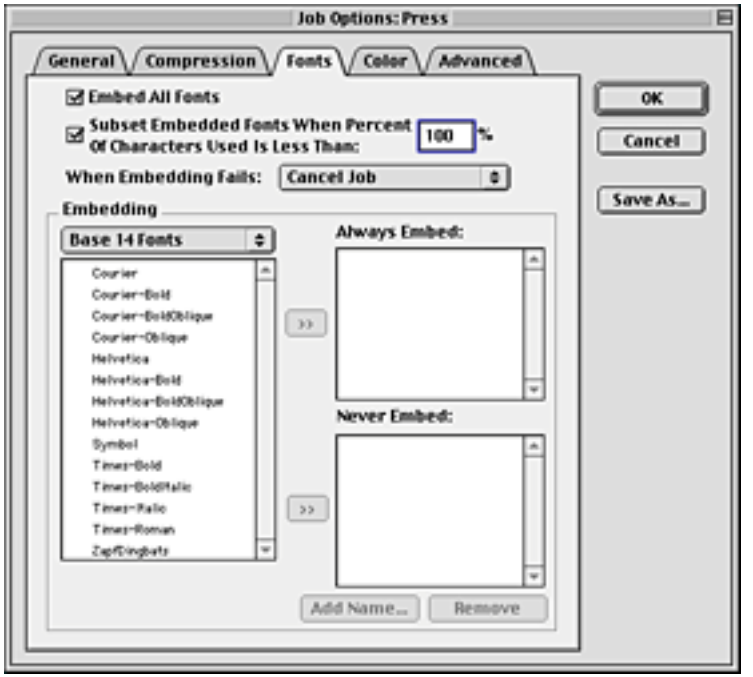
Let's see how it works.

Next Page ->

Background: Embedding Fonts

Fonts are embedded in your PDF file by Distiller. Among Distiller's Job Options is a Fonts panel that lets you specify that fonts should be embedded.

It's pretty simple: just go to *Settings* > *Job Options,* select the *Fonts* tab, and make sure the *Embed All Fonts* button is selected.



If you want to make your PDF file more compact, you can click the *Subset Embedded Fonts* button; for each font, Distiller will embed only the characters that are actually used in the PDF file. (If you don't subset your fonts, Distiller will embed each font in its entirety, including all the characters that are not used in this document.) Be careful about subsetting your fonts; it makes the PDF file smaller, but it also interferes with your ability to later edit the text in the file.

There's more to say about subsetting fonts, but we don't have space here, unfortunately. (Maybe a future *Journal* article?)

Next Page ->

If a Font Isn't Embedded

If your PDF file uses Optima, say, but Optima isn't included in the PDF file, Acrobat will need to decide how to display that text. It will solve the problem one of two ways:

Use a System Font

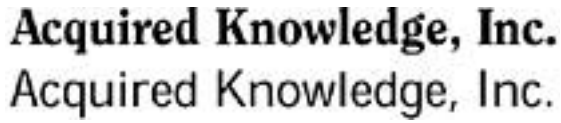
If Optima is installed on the computer being used to view the PDF file (the "viewing computer"), then the Acrobat viewer will use the viewing computer's Optima. If that happens to be the same version of Optima you used to lay out the document, then the text will display and print exactly as in your original document.

On the other hand, if the viewing system's Optima is different from yours (from a different manufacturer, perhaps), the document may display and print with a variety of metrics-related errors: flush right text becomes slightly ragged, previously-centered text is now off-center, etc. (The text won't reflow, if that helps any.)

Create a Stand-In

If Optima is neither embedded in the file nor installed on the viewing system, then the Acrobat viewer will use Adobe's Multiple Master technology to create a "stand-in" for that font. The character widths and other metric characteristics of the stand-in font will match those of the original; the intent is to preserve overall "feel" of the document.

This is quite clever, although the result is not a good replacement for the original font. For example, at right we have a sample of Clearface Heavy and, beneath it, the stand-in font created by Acrobat. The metrics (character widths, X height, cap height, descenders) match the original, but the character shapes are not very much like the originals.



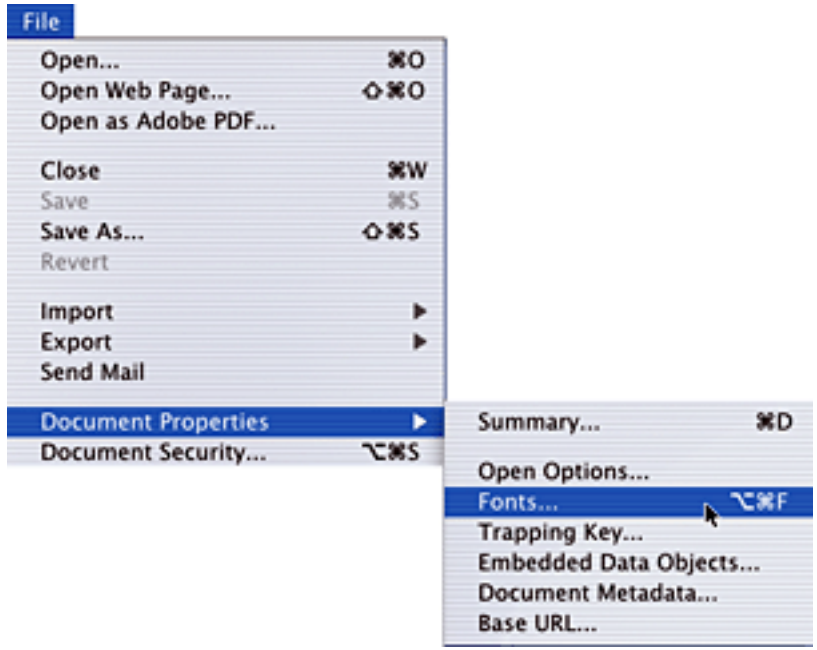
Next Page ->

Document Fonts

In Acrobat 4, this dialog box is named *Font Info.* You get to it by selecting *File>Document Info>Fonts.*

Clearly, it is important that your PDF files contain the definitions of the fonts they need.

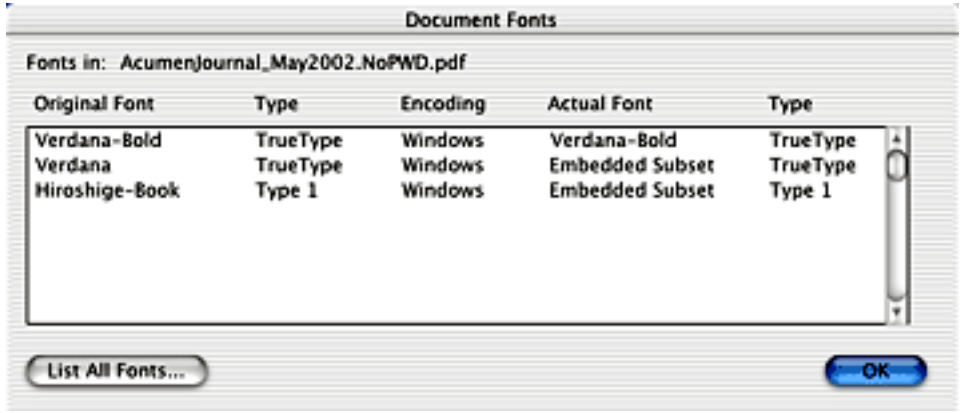
Acrobat provides a tool that lets you determine whether any of the fonts in your PDF unembedded. If you select *Fonts* > *Document Properties* > *Fonts* in the File menu, Acrobat will present you with the *Document Fonts* dialog box.



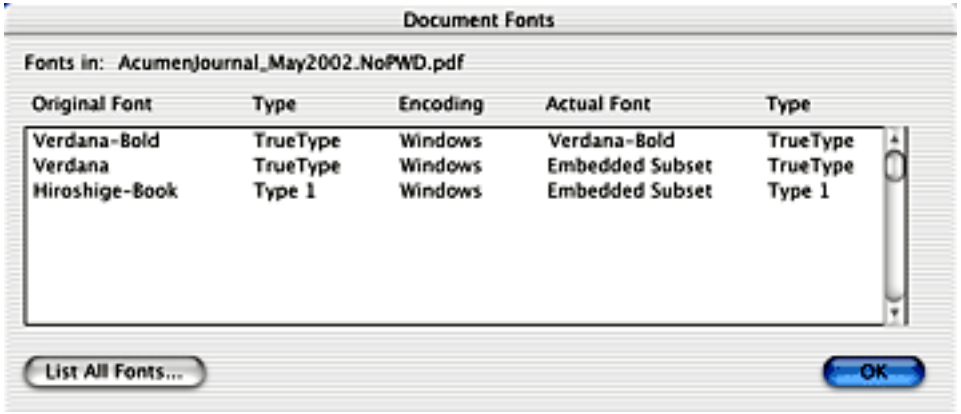
This dialog box describes all of the fonts used in the PDF file and will let you determine if any of the fonts used by your PDF file are missing.

Let's take a look at what, exactly, the *Document Fonts* dialog box tells us and then we'll examine its limitations.

Next Page ->



What's Here This dialog box supplies five pieces of information for each font used in the document, of which only one is particularly useful, diagnostically.



Original Font/Type This is the name and type of the font used in the original layout. The font type will be one of the following:

- *Type 1* – The most commonly-used PostScript font type.
- *TrueType* – A TrueType font in native format. (These are also called "Type 42" fonts.)
- *Type 3* – This is the native PostScript font format. These days, it is used primarily for TrueType fonts that have been converted to a bitmap font.

Encoding This is the character encoding used by the font, that is, the set of character codes that are used internally to represent text characters. This can be a variety of things, including MacRoman, Windows, and Custom.

This sounds more important than it really is. In particular, the encoding of a font will not prevent it from being displayed and printed on any computer platform. The interpretation of the font's encoding is built into the PDF file.

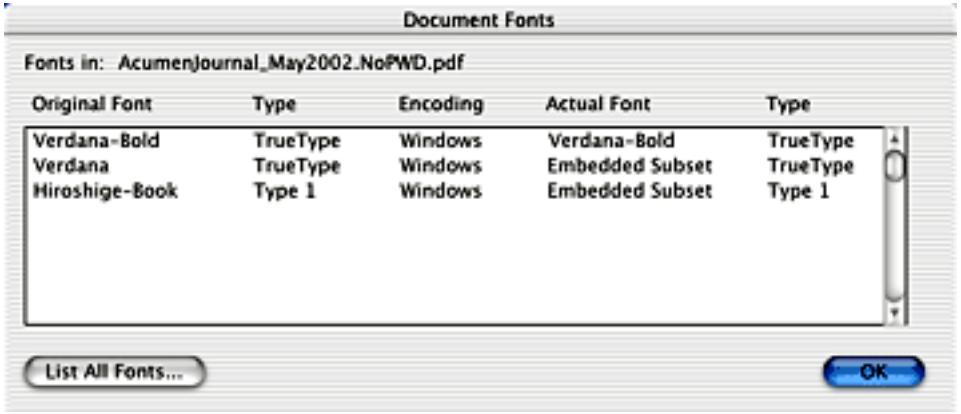
Next Page ->

Acumen Journal: Acrobat User 6

Actual Font/Type

These columns describe the font that is actually used to display and print the PDF file on the viewing computer.

The Actual Font column is the important one; it lets you head off missing fonts. Let's talk about it in detail.



The *Actual Font* Column

The *Actual Font* column lists how fonts are being represented on the viewing computer. Each font will have one of four entries, each implying something that may be a problem in the display and printing of this PDF file.

Same as Original Font (Bad)

If the Actual Font is the same as the Original Font, this font is being displayed with the viewing system's version of the font.

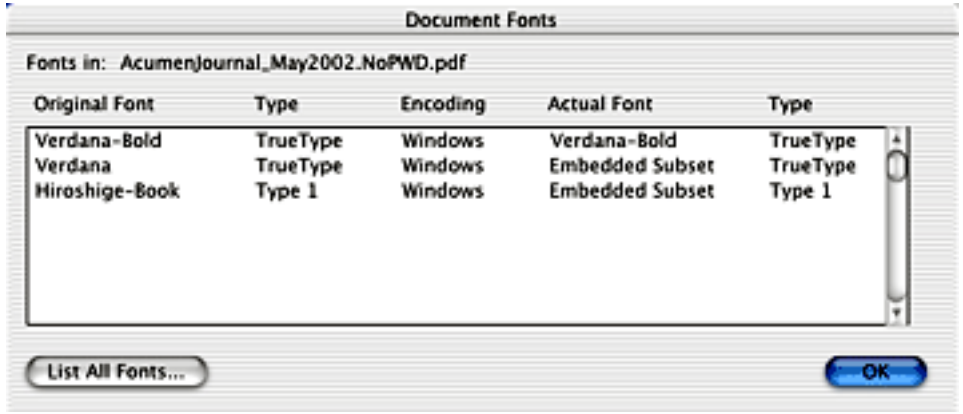
This is a concern, since it means the font *isn't* embedded in the PDF file. It looks alright for the moment, because the font happens to be installed on the viewing computer and that is what our viewer is using to display the text.

However, if we take this document to another computer that doesn't have the font installed, the viewer will use a stand-in font in its place. This will not look good. You should regenerate the PDF file with the fonts embedded.

Next Page ->

Embedded (good)

The font is embedded in the PDF file and that is what the viewer is using to display the text. This is good; an embedded font is a font we don't need to worry about.



Embedded Subset (good?)

The font is embedded in the PDF file as a subset and this is what is being used to display the text. Here, too, we don't have much to worry about. The font as displayed and printed is exactly what was used to lay out the original document.

There is only one possible problem: text in a subsetted font is functionally impossible to edit in the PDF file. You will be unable to modify the text that uses this font. If editability is important to you, you should regenerate the PDF file, turning subsetting off.

AdobeSansMM
AdobeSerifMM
(¡Muy malo!)

Here's where we have troubles. These are the Multiple Master fonts that Acrobat uses to construct stand-ins fonts. One of these two names will appear in the *Actual Font* column if everything has gone wrong: the original font was not embedded in the PDF file and it is not installed on the viewing computer system. What you are seeing on-screen (and what will print) is the stand-in font. Not what you had in mind.

Go back, regenerate the PDF file, and make sure the fonts are embedded this time.

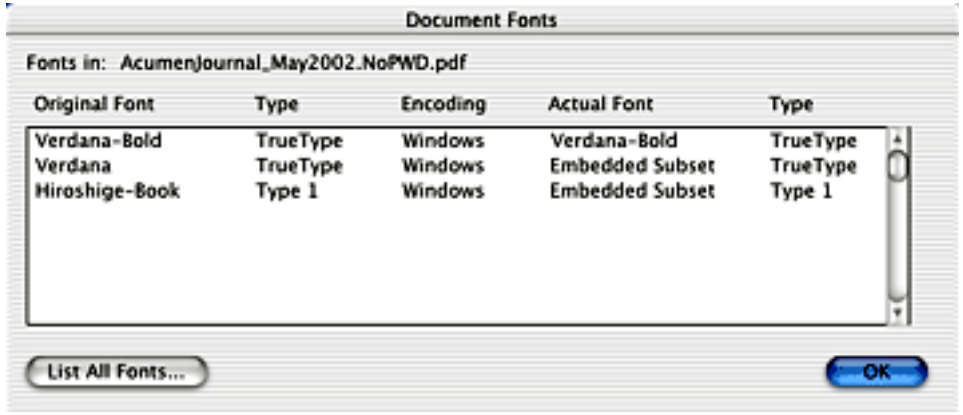
Next Page ->

Acumen Journal: Acrobat User 8

Limitations

The *Document Fonts* dialog box can be extremely useful in heading off problems associated with unembedded fonts. Unfortunately, it is subject to a significant limitation: it only shows "Actual Font" information for fonts it has actually tried to draw on screen.

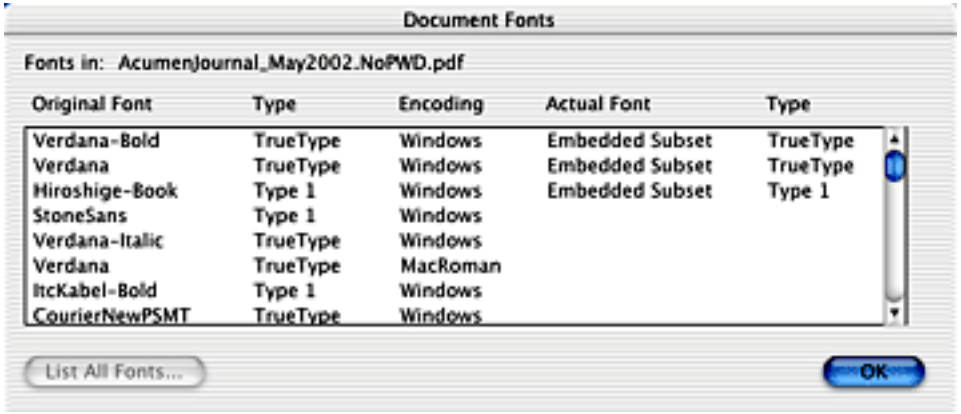
If you open a PDF file and immediately bring up *Document Fonts,* the dialog box will show information for only the first page of the document, as at right. Acrobat doesn't look ahead in the document, so it doesn't yet know what fonts are in there, let alone how it will draw them.



Next Page ->

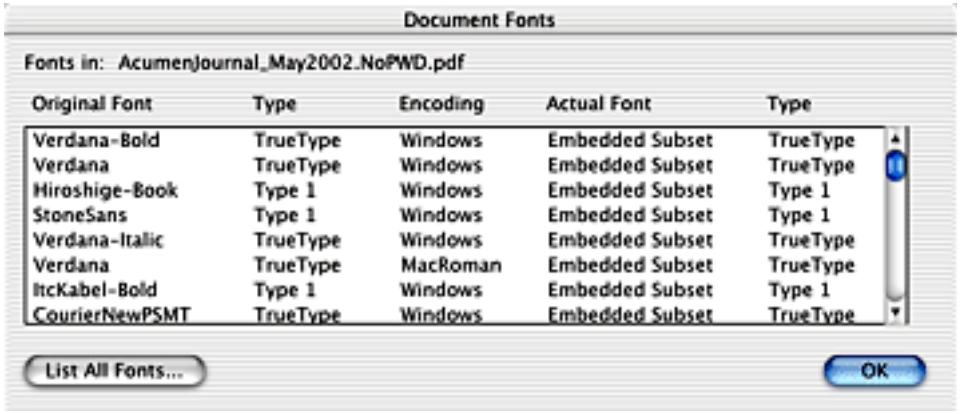
List All Fonts

If you click on the *List All Fonts* button, Acrobat will scan the PDF file and add to the "Original Font" column all the fonts used in the document. The "Actual Font" column will still be blank, because Acrobat doesn't decide how to display a font until it is actually called to paint the text's characters on the screen.



The Problem

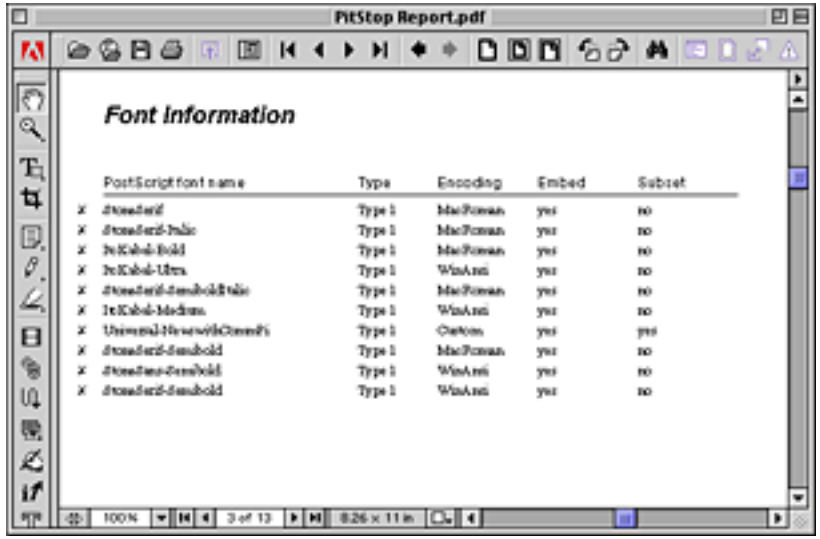
Here's the painful bit: to see the Actual Font entry for all the fonts — and, therefore, determine if any of the fonts are unembedded — you must first page through the document, giving Acrobat time to draw *each and every page,* and then bring up the *Document Fonts* dialog box.



Only then can you tell whether any of the fonts are missing.

Next Page ->

Use Preflight Software



Running through your PDF file one page at a is a pretty boring process if the PDF file is even a little long; this places a severe practical restriction on the usefulness of the *Document Fonts* dialog box as a routine diagnostic tool. Still, it is perfectly useful on short PDF files.

For longer files and for routine preflighting in a production environment, you should invest in a preflight utility: Enfocus' *PitStop* and Markzware's *Flightcheck* come immediately to mind. Any worthwhile PDF preflight tool will give you a list of unembedded fonts in no time.

Return to Main Menu

Skipping Blocks of Code with *SubFileDecode*

Among the many things that keep my social life simple is the delight I take in some of the less-used mechanisms in the PostScript language. At parties and dinners, I tend to get inappropriately fervent in my discussions of composite fonts and automatic stroke adjustment.

This is why most of our dinner invitations are addressed to my wife.

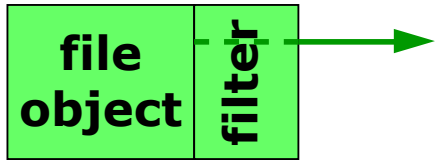
This month and next we'll discuss something you will have heard me hold forth on if you have taken the Advanced PostScript class: the *SubFileDecode* filter. This under-appreciated filter has a variety of useful applications, including the ability to skip past large blocks of PostScript code and to restrict the effect of PostScript errors.

This month, we discuss what SubFileDecode does and see how to use it to conditionally bypass large chunks of PostScript code. This technique is very useful for including multiple versions of your procedure definitions (to support different PostScript LanguageLevels, for example) and executing one or the other, depending on circumstance.

Let's start by reviewing how filters work in PostScript Levels 2 and 3.

Next Page ->

PostScript Filters



You probably remember filters from your PostScript classes. A filter is a device you can attach to a PostScript file object. Having attached the filter to a file, you can read data from or write data to that file through the filter; any data passing through the filter is altered in some way: converted between ASCII to binary, compressed, or uncompressed.

PostScript *Encode* filters take raw data and convert it to another form: usually convert it to ASCII or compress it. *Decode* filters take the encoded data and convert it back to its original form.

At right is a list of all of the Decode filters currently available in PostScript. These fall into three broad classes:

- *Transmission filters* convert ASCII-encoded binary back to the original binary data.
- *Compression filters* uncompress the data passing through them.
- *Dummy filters* do not change the data passing through them.

PostScript Decoding Filters

Transmission Filters	
ASCIIHexDecode	ASCII85Decode
Compression Filters	
LZWDecode	CCITTFaxDecode
RunLengthDecode	DCSDecode
FlateDecode	
Dummy Filters	
ReusableStreamDecode	SubFileDecode

Each PostScript Transmission and Compression filter also has an Encode version.

Next Page ->

The *filter* Operator

You attach a filter to a file with the *filter* operator.

```
fileobj <params> /Filtername filter  =>  filtered-fileobj
```

This operator takes from the stack the name of the filter, the file object to which you want to attach the filter, and, in between them, whatever parameters are required by the filter. (Most filters have no parameters.) The operator attaches the filter to the file object and returns a *filtered fileobject.*

A filtered fileobject looks to PostScript exactly as though it were a regular file object. Within your PostScript program, you read data from or write data to this object just like any other file. However, anything you read from this file will be passed through the attached filter and modified before being handed on to the PostScript operator that does the reading.

You can hand a filtered fileobject to any PostScript operator that takes a fileobject as its argument: *readstring, readline,* etc.

Next Page ->

Acumen Journal: PostScript Tech 14

For Example… For example, the following code reads a kilobyte of LZW-compressed binary data.

```
/lzwSource currentfile /LZWDecode filter def
/buffer 1024 string def

lzwSource buffer readstring
... LZW-compressed data goes here ...
```

Our data source is *currentfile* with the LZWDecode filter attached. When the *readstring* operator reads data from this filtered fileobject, the data will be read from *currentfile,* un-LZW-compressed, and then placed into the string *buffer.* This will continue until *buffer* is filled or we hit end-of-file.

Next Page ->

SubFileDecode

SubFileDecode is a dummy filter; it doesn't alter the data passing through it. At first glance this seems pretty useless, but, in fact, it is astonishingly handy. By attaching SubFileDecode to *currentfile,* you can divide your PostScript stream into a series of logical "subfiles." This, in turn, allows you to do some remarkably useful things.

Attaching *SubFileDecode*

When you attach SubFileDecode to a file, you need to provide a pair of parameters to the *filter* operator:

```
fileobj  count  (EOFString)  /SubFileDecode  filter
```

- *EOFString* is the string that defines the end of the subfile. The filtered file object will register end-of-file when this string of characters passes through it.
- *Count* is the number of instances of *EOFString* that should be ignored before returning EOF. If *count* is zero, then the first instance of *EOFString* will cause EOF.

Let's look at what you can do with this filter.

Next Page ->

Acumen Journal: PostScript Tech 16

Skipping Code

SubFileDecode is an excellent tool for conditionally skipping long blocks of PostScript code. This is particularly useful if your PostScript stream has two or more versions of its procedure definitions, usually to accommodate different versions of PostScript.

Without SubFileDecode

Without SubFileDecode, you would do something like this:

```
isLevel3        % This is a boolean value, let's assume
{     ...
      ... Level 3 definitions ...
      ...
}
{     ...
      ... Level 2 definitions ...
      ...
} ifelse
```

This is perfectly appropriate if you have only a couple of definitions in your ProcSet. However, if your alternative ProcSets are large, you are faced with a memory problem: both the Level 2 and Level 3 procedures must be constructed and placed on the stack before you call *ifelse.* Thus, regardless of which set of definitions you actually use, you must store both ProcSets in VM before choosing between them.

The memory manager will eventually garbage collect the *ifelse* procedures, but this is still bothersome; garbage collection can be very slow. It's also esthetically unpleasing; I dislike creating things in memory (particularly big things) that I'm never going to use.

Next Page ->

With SubFileDecode

So how do we do this without a memory problem? With *SubFileDecode,* of course:

```
currentfile 0 (*EOF*) /SubFileDecode filter
cvx exec                              % Execute currentfile thru' the filter
isLevel3 not                          % Do we not have a Level 3 interpreter?
{ currentfile flushfile } if          %    ...then flush to end-of-subfile
...                                   % Otherwise continue Lvl 3 proc def's
... Level 3 procedure defs
...
*EOF*                                 % End of subfile


                                      % Do it again for the Level 2 proc's
currentfile 0 (*EOF*) /SubFileDecode filter
cvx exec
isLevel3 { currentfile flushfile } if
...
... Level 2 procedure defs
...
*EOF*
...
... Rest of PostScript program
...
```

Let's step through this in detail.

Next Page ->

The Code in Detail

```
currentfile 0 (*EOF*) /SubFileDecode filter
```

Here we attach the SubFileDecode filter to *currentfile,* our currently-executing PostScript stream. This creates a new subfile what will end the first time the text "*EOF*" passes through it. (Note we are ignoring 0 instances of "*EOF*.")

The *filter* operator returns a filtered fileobject representing the subfile.

```
cvx exec
```

We convert the filtered fileobject (left on the stack by *filter)* to executable and then execute it with *exec.* Everything from this point to **EOF** will be executed through the SubFileDecode filter.

```
isLevel3 not
{ currentfile flushfile } if
```

We want to skip over the Level 3 ProcSet if we are don't have a Level 3 interpreter. To do this, we examine the *isLevel3* boolean (presumably set earlier); if it is false, we flush to the end of currentfile. This is not as alarming as it seems; remember that at this moment, our currently executing stream (returned by *currentfile)* is the SubFileDecode subfile. Flushing currentfile causes the interpreter to skip to just after **EOF*.*

```
*EOF*
```

This is the end of the subfile. The interpreter resumes executing our PostScript stream directly (not through *SubFileDecode)* immediately after this marker.

Next Page ->

Acumen Journal: PostScript Tech 19

```
currentfile 0 (*EOF*) /SubFileDecode filter
cvx exec
isLevel3 { currentfile flushfile } if
...
*EOF*
```

The second block conditionally defines the Level 2 ProcSet, skipping over the code if our interpreter supports LanguageLevel 3.

So what does this get us?

Using SubFileDecode to conditionally execute our PostScript code means we do not need to construct (and, therefore, store in VM) a procedure body holding the unused ProcSet. More than anything else, this saves memory; it also saves an insignificant bit of processing time (the scanner doesn't have to convert the procedure body contents to PostScript objects). Transmission time will be nearly identical, of course.

Saving VM is always a virtue.

Next Month: Isolating Errors

Next month we'll look at a couple of other things you can do with SubFileDecode. In particular, we'll see how to isolate the effect of a PostScript error so that it doesn't kill the entire rest of the PostScript stream. (There is nothing more annoying than having a PostScript error in page 2 of a 20,000-page print run kill the remaining 19,998 pages; we can fix that!)

Return to Main Menu

Schedule of Classes, June – August, 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	July 29 – Aug 2	
Advanced PostScript	June 24 – 28	August 12 – 16
PostScript for Support Engineers	August 5 – 9	
Jaws Development	Sept 30 – Oct 3	

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

These classes are taught quarterly in Costa Mesa, California, and on corporate sites. Clicking on a course name below will take you to the class description on the Acumen Training web site.

Acrobat Essentials	Jun 20 (1/2-day, morning)
Interactive Acrobat	
Creating Acrobat Forms	Jun 20 (1/2-day, afternoon)
Troubleshooting with Enfocus' PitStop	June 21 (Full day)

Acrobat Class Fees

Acrobat Essentials and *Creating Acrobat Forms* (1/2-day each) cost $180.00 or $340.00 for both classes. *Troubleshooting With PitStop* (full day) is $340.00. In all cases, there is a 10% discount if three or more people from the same organization sign up for the same class.

Registration ->

Return to Main Menu

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

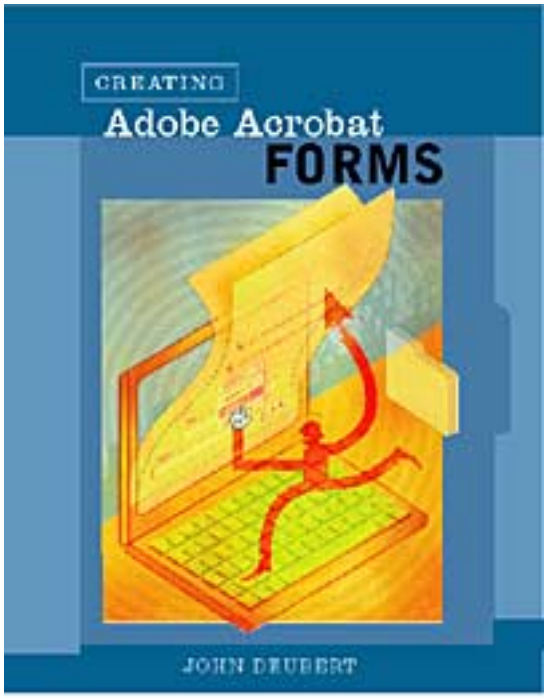
Back issues of the Acumen Journal are available at the Acumen Training website: www.acumenjournal.com/AcumenJournal.html

Return to First Page

What's New at Acumen Training?

Creating Acrobat Forms is Out!

This new book Adobe Press book by John Deubert is a beginner-to-intermediate guide to the creation and use of forms in Adobe Acrobat. Presuming no knowledge on the part of the reader beyond basic Acrobat use, this book steps the reader through the process of creating a form in Adobe Acrobat. It details the characteristics and behavior of each of the Acrobat form field types, and outlines how to submit the data a form has collected to a remote server for processing.



If you are looking for an easily-read, concise, accurate manual that will teach you how to create Acrobat forms for your organization, you will find this book entirely satisfactory.

Creating Adobe Acrobat Forms is an Adobe Press book published by Peachpit Press and is available in all Better Bookstores, as well as on-line sources, such as Amazon.com.

Click here for a list of chapters on the Acumen Training website.

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?

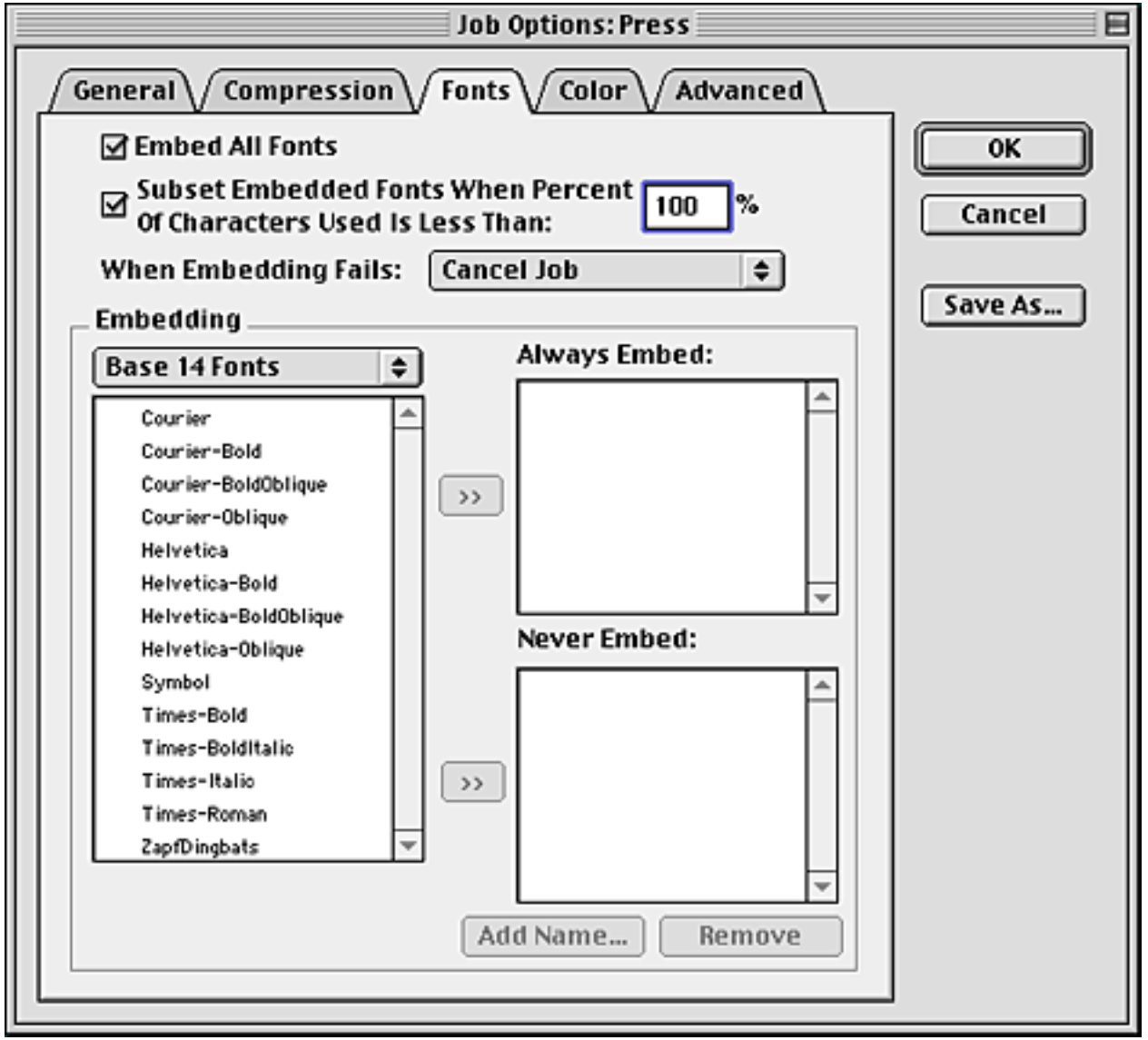
Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

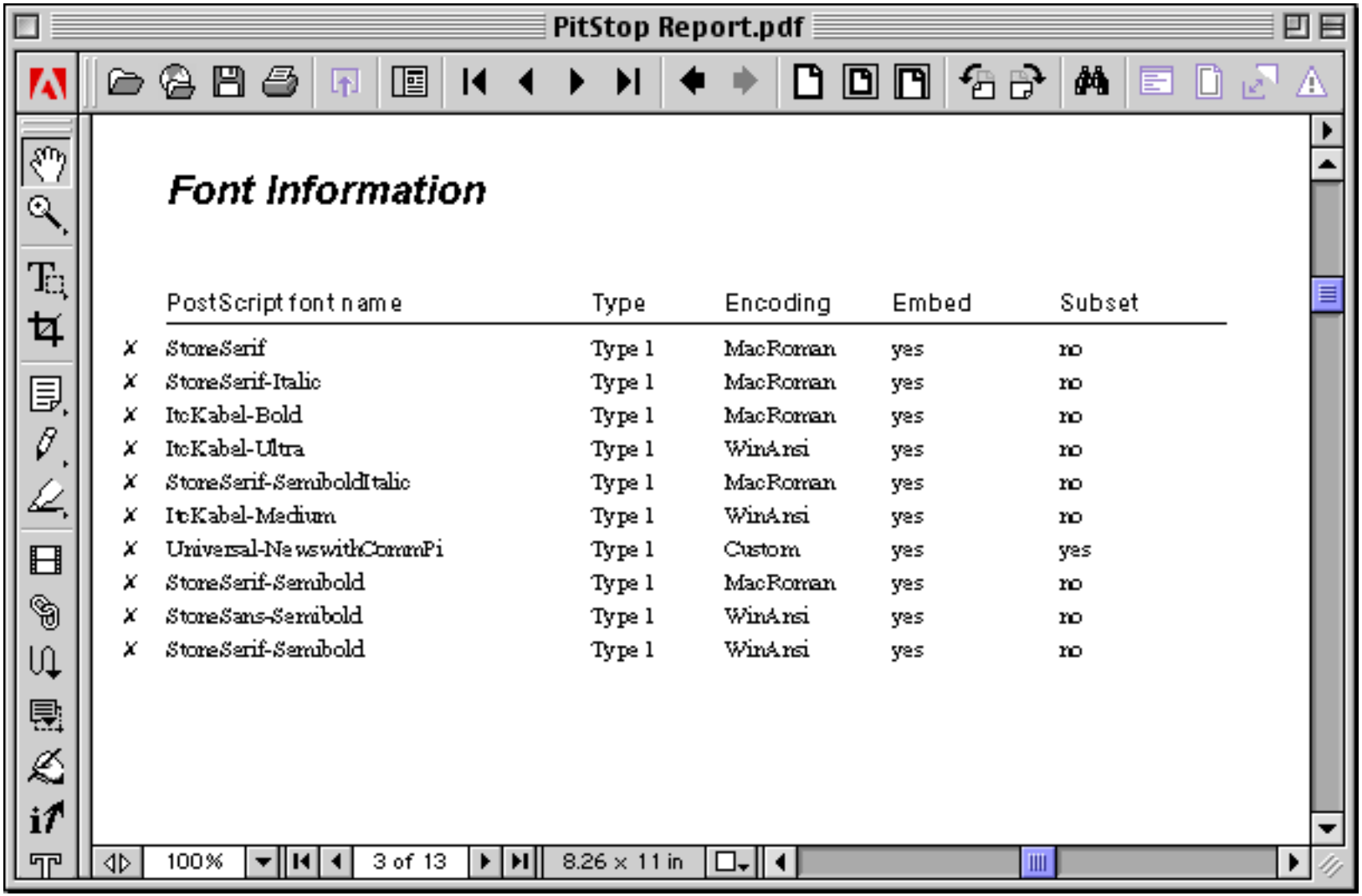
Questions and Answers. Do you have any questions about Acrobat, PDF or PostScript? Feel free to email me about. I'll answer your question if I can. If enough people ask the same question, I can turn it into a Journal article.

Please send any comments, questions, or problems to:

journal@acumentraining.com

Return to Menu


Job Options: Press
General
Compression
Fonts
Color
Advanced
Embed All Fonts
Subset Embedded Fonts When Percent Of Characters Used Is Less Than:
100
%
When Embedding Fails:
Cancel Job
OK
Cancel
Save As...
Embedding
Base 14 Fonts
Courier
Courier-Bold
Courier-BoldOblique
Courier-Oblique
Helvetica
Helvetica-Bold
Helvetica-BoldOblique
Helvetica-Oblique
Symbol
Times-Bold
Times-BoldItalic
Times-Italic
Times-Roman
ZapfDingbats
>>
Always Embed:
>>
Never Embed:
Add Name...
Remove


PitStop Report.pdf
Font Information
PostScript font name
Type
Encoding
Embed
Subset
StoreSerif
Type 1
MacRoman
yes
no
StoreSerif-Italic
Type 1
MacRoman
yes
no
ItcKabel-Bold
Type 1
MacRoman
yes
no
ItcKabel-Ultra
Type 1
WinAnsi
yes
no
StoreSerif-SemiboldItalic
Type 1
MacRoman
yes
no
ItcKabel-Medium
Type 1
WinAnsi
yes
no
Universal-NewswithCommPi
Type 1
Custom
yes
yes
StoreSerif-Semibold
Type 1
MacRoman
yes
no
StoreSans-Semibold
Type 1
WinAnsi
yes
no
StoreSerif-Semibold
Type 1
WinAnsi
yes
no
100%
3 of 13
8.26 x 11 in