Table of Contents

Acumen Training

Password Protecting Acrobat Files

I'm surprised we haven't talked about this question before in this periodical: How do you password-protect an Acrobat document, preventing readers from modifying it? This is a common need for anyone who distributes copyrighted documents in Acrobat format. As an obvious example, the *Acumen Journal* is password protected so that you, the reader, can't change the document's contents. (Sorry.)

As it turns out, Acrobat allows you to easily attach to a PDF file a password that a reader must supply in order to open that file and an additional password that is needed to make other kinds of changes to the file: print it, modifying it, fill out form fields, etc.

This month, we'll step through the process of attaching these passwords to a file.

Next Page ->

Permissions and Passwords

You can associate up to two passwords with a PDF file:

- A *User Password* required to open the PDF file. You may use this password to keep Unauthorized Personnel from looking at your document.
- A *Master Password* that the reader must know in order to change restrictions you have placed on the document. There are four of these restrictions, all pretty self-descriptive:
  - *No printing:* the user can't print the document.
  - *No changing the document:* the user can't change the contents of the document. The user *can* fill out form fields and attach annotations to the file.
  - *No content copying or extraction, Disable accessibility:* This grays-out all of the tools that would let a reader copy text or graphics from the PDF file into another document. It also turns off the "accessibility" features that provide keyboard access to all dialog items, etc.
  - *No adding or changing comments and form fields:* Readers won't be able to fill out form fields or annotate the document.

These two passwords are independent: you may specify one, both, or neither.

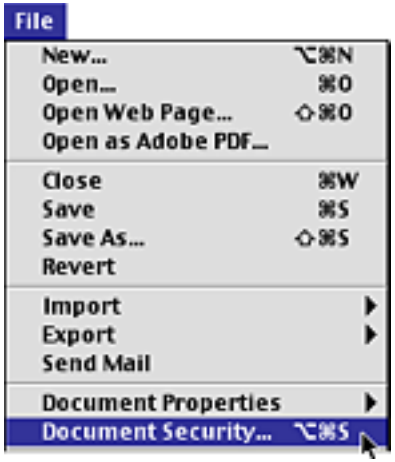
Next Page ->

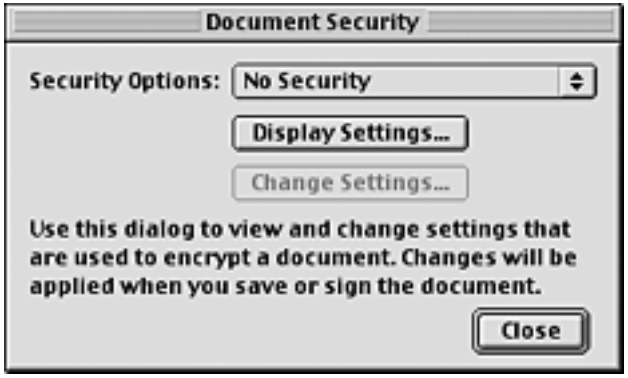
Specifying Passwords

To password-protect an Acrobat document, start with the document open in Adobe Acrobat (you must use the full Acrobat for this, not the Reader), then do the following:

1. Select *File>Document Security…*

   Acrobat will present you with the *Document Security* dialog box (below, right).





Next Page ->

2. Select *Acrobat Standard Security.*

   The set of choices presented to you in the *Document Security's* pop-up menu may be augmented by security plug-ins you may have installed. The default security methods supplied by Acrobat are *Acrobat Standard Security* and *Acrobat Self-Sign Security.*

   Self-sign security allows only people with certain electronic signatures to open the file. This allows you to limit access to only people you have specifically okayed. (For information about Acrobat Self-Sign Security, see the October and November 2001 issues of the *Acumen Journal* or, for a fuller treatment, my book *Creating Acrobat Forms* by Adobe Press.)

   In this article, we discuss Acrobat Standard Security.

3. Click on the *Change Settings…* button.

   Acrobat presents you with the *Standard Security* dialog box (at right).

Next Page ->

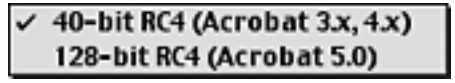
4. Select the restrictions you want to place on the document, if any.

   The lower half of the *Standard Security* dialog box lets you select which permissions you want to withhold from the readers of your document. Select as many of the four checkboxes you wish.

5. Specify one or both of your passwords.

   You may specify a password needed to open the document and/or one needed to override the other restrictions. (You will also need this second password if you want to completely remove passwords from your document.

6. Select what level of encryption you want applied to the contents of your PDF file.

   You may choose between 40- and 128-bit encryption. The 128-bit encryption is more secure, but is not readable by Acrobat 4. If you expect some of the people who read your file to have older versions of Acrobat, you should select 40-bit encryption.

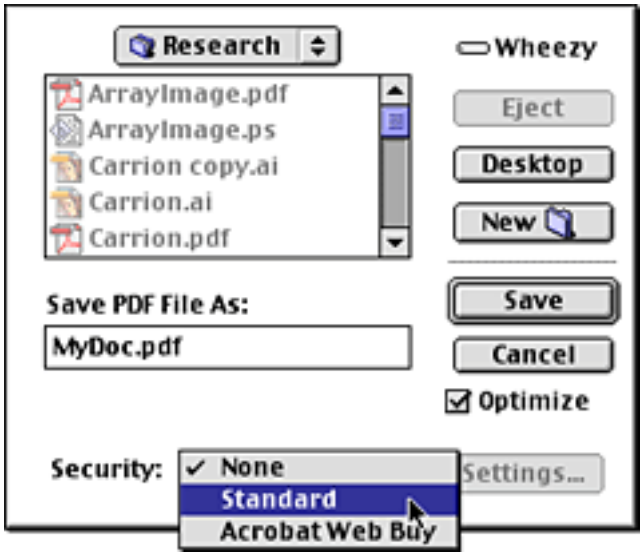
7. Click the *OK* button.

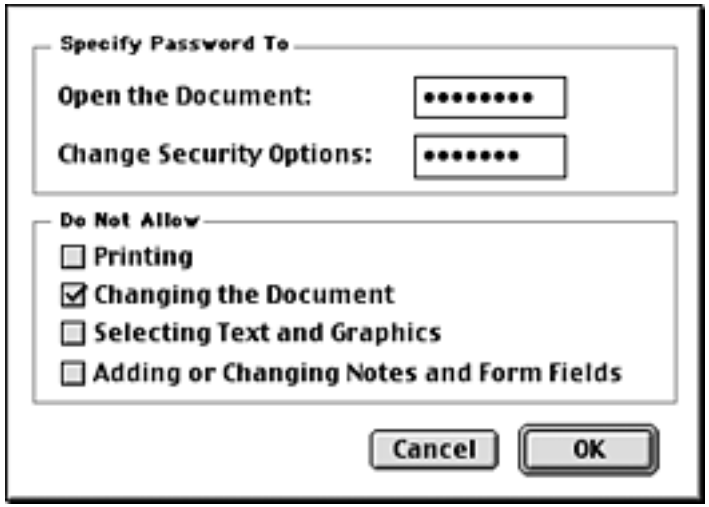
That's it; your file is now password-protected.

Next Page ->

Acrobat 4 If you are still using Acrobat 4 (I still do, sometimes, to avoid oddities in Acrobat 5), the set of permissions and the two passwords are identical to those in Acrobat 5, though the way you get to them is different. In Acrobat 4, you add password protection to your document from the *Save as…* dialog box.

When you select *File>Save as…,* the *Save-File-As* dialog box has a pop-up menu that allows you to specify security that should be applied to the saved file. If you select *Standard,* Acrobat presents you with a dialog box that allows you the same choices as are offered by Acrobat 5.





Next Page ->

Security Plug-ins

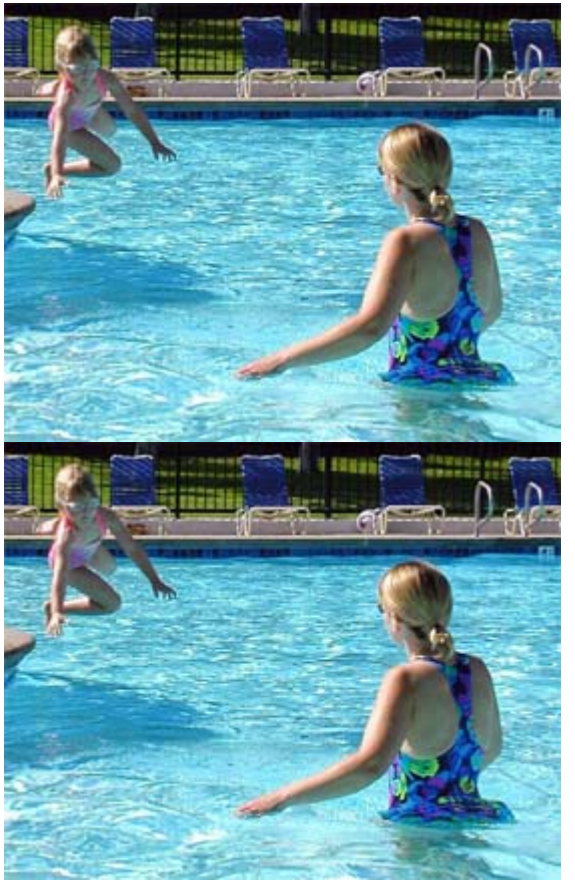
With Acrobat 5, Adobe made the security mechanism modular and extensible, allowing third parties to create Acrobat plug-ins that add new types of excryption to Acrobat. There are several companies that have written plug-ins that advantage of this, including *SecurSign* by Appligent and *Sign-it* by CIC.

If you are curious about additional security plug-ins for Acrobat, I suggest you wander over to *PlanetPDF.com* and look at their list of Acrobat plug-ins.

Return to Main Menu

Using Images in Forms, Part 2

Last month, we started a discussion of how to create PostScript forms that contain scanned images. We saved the image data in a file on the RIP's hard disk and then created a PostScript form whose *PaintProc* procedure made a call to the *image* operator that read the data from that image data file.



This worked very well; we used it to turn an image into a form and then executed the form twice, producing the results at right. Unfortunately, last month's technique requires the RIP have a hard disk directly available to it, something that is not true of many RIPs.

This month, we shall examine a second way of using an image from within a form: using the *ReusableStreamDecode* filter to read the image data into memory as a virtual file.

Next month, we'll look at a third method: creating an array of strings that together contain the image data.

If you haven't read last month's article, you should do so now; I'm going to assume it is still fresh in your memory.

Next Page ->

Acumen Journal: PostScript Tech

ReusableStream-Decode

ReusableStreamDecode is a PostScript filter; its use in our example here will allow us to read the entire image data into VM and then read it repeatedly as though it were a repositionable file. In effect, we create a virtual file in RAM containing our image data; our call to the image operator will then read data from that file, pretty much exactly as we did last month.

PostScript Filters

Like all PostScript filters, *ReusableStreamDecode* is a dingus you attach to a PostScript data source, usually a file object. The result is what I call a "filtered-fileobject." This looks to PostScript exactly like a normal read-access file object: you can read data from it as you would from any file; however, data read from or written to this file object is passing through the filter and changed in a manner specific to that filter. Depending upon the filter, the data will be compressed, uncompressed, converted from binary to some flavor of ASCII or *vice versa,* or not changed at all.

The filter operator

You attach a filter to a PostScript file with the *filter* operator:

```
fileobj  params-if-any  /FilterName  filter  =>  filtered-fileobj
```

The arguments are: the fileobject to which the filter should be attached; parameters, if any, specific to the filter; the name of the filter you want. I shall refer you to your PostScript notes or the PostScript Language Reference Manual for a complete list of the filters available; the name we shall be using here is */ReusableStreamDecode.*

You should remember this stuff, at least vaguely. We briefly discuss filters in the PostScript Foundations and Support Engineers classes. We go into them in depth in the Advanced PostScript class.

Next Page ->

Acumen Journal: PostScript Tech 10

ReusableStreamDecode

The *ReusableStreamDecode* filter is an interesting beast. When you attach it to a file object (upon execution of the *filter* operator), PostScript immediately reads the entire contents of that file into VM, returning a filtered-fileobject that represents that data in memory. That filtered-fileobject behaves in every way as though it were an actual, read-access, repositionable file; in particular, you can read data from it and rewind it to the beginning so you can read it again.

Thus,

```
currentfile /ReusableStreamDecode filter
```

will read the entire rest of our PostScript stream into memory and return a fileobject representing that data.

Reading *all* of the remaining PostScript stream into VM is not particularly useful; it leaves us with no remaining PostScript code to operate on our virtual file. Usually, you attach *ReusableStreamDecode* with parameters that one way or another specify when to stop reading the input stream into memory and, therefore, to resume executing it as PostScript.

A complete discussion of how to use *ReusableStreamDecode* is way beyond what we have time and space for here; take the *Advanced PostScript* class for the full story. Here, we shall talk only about how to use this filter in our image-in-a-form project.

Next Page ->

Acumen Journal: PostScript Tech 11

The PostScript Code

Here's the *ReusableStreamDecode* version of our form. This assumes our image data is ASCIIHex encoded; we'll talk about how to handle some of the other possibilities later.

Read data into VM

```
/ImageData                          % The name of our eventual "virtual file"
currentfile                         % We'll attach the filter to currentfile
<< /Filter /ASCIIHexDecode >>       % Run data through this filter (see text)
% Now attach the filter and start reading data:
/ReusableStreamDecode filter
2c192d200f1f2213182319181d14171914181b161d121117121212141611
16191211140d0e0e0e13121716131c0d0b161517240d111c12151c13171a
... Whole heap o' ASCIIHex-encoded image data ...
7fcbd986d4e188d4e286d4e18ad6e48ddbe88ad6e48fddea98e4f29debf8
a1edfba2f0fda6f2ffa9f5ffaef6ffaef7ffaef7ffaef7ffaef7ffaef7ff
>                   % End-of-data marker for ASCIIHexDecode data
def                 % Stack: /ImageData fileobj => ---
```

This month's sample programs are on the Acumen Training Resources page as *ImageForm2.zip.* This file is in the zip file as *ReusableStream.ps.*

Next Page ->>

Create the form

```
/JumpForm <<
     /FormType 1                          % FormType is always 1
     /BBox [ 0 0 278 219 ]                % The form prints image at 0,0
     /Matrix [ 1 0 0 1 0 0 ]              % Translate by 0,0; scale by 1,1

     /PaintProc                           % Here we draw our form
     {    pop                             % Discard the form dict. argument
          ImageData 0 setfileposition     % "Rewind" our image data
          /DeviceRGB setcolorspace        % We have an RGB image
          278 219 scale                   % Width and height of the data
          <<   /ImageType 1
               /Width 278                 % Image is 278 samples across...
               /Height 219                % ...and 219 scanlines hight
               /BitsPerComponent 8        % 8 bits each of R, G, and B
               /ImageMatrix [ 278 0 0 -219 0 219 ]    % [ w 0 0 -h 0 h ]
               /DataSource ImageData      % The rewound data-in-VM
               /Decode [ 0 1 0 1 0 1 ]    % Map data 00…ff into color 0…1
          >> image                        % Call the image operator
     } bind
>> def
```

Use the form

```
JumpForm execform   % Here the PaintProc gets executed
0 219 translate
JumpForm execform   % Here the form is rendered from the cache
```

Next Page ->

Step by Step

```
/ImageData
```

We start by placing on the stack the name *ImageData,* which is the name we shall eventually associate with our virtual file in VM.

We follow this with a call to the *filter* operator that finds three arguments on the stack:

```
currentfile
```

This is the object to which we shall attach the *ReusableStreamDecode* filter. We are going to read image data from our input stream and place it in VM.

```
<< /Filter /ASCIIHexDecode >>
```

This is a dictionarythat contains parameters for our *ReusableStreamDecode* filter. In this case, we are supplying the name of another filter, *ASCIIHexDecode.* The data read from *currentfile* will be passed through this filter before being stored in VM. Thus, the data we read will be converted from ASCIIHex to binary on its way to VM; this will halve our data's footprint in memory. As a by-product, this will also allow us to specify where the image data ends and regular PostScript code resumes; we'll see this in a moment.

```
/ReusableStreamDecode filter
```

Finally, we place the name */ReusableStreamDecode* on the stack and execute *filter.* The *filter* operator immediately starts reading data from *currentfile,* passing the data through the *ASCIIHexDecode* filter and placing the resulting binary data into VM.

Next Page ->

```
2c192d200f1f2213182319181d14171914181b161d121117121212141611
...
a1edfba2f0fda6f2ffa9f5ffaef6ffaef7ffaef7ffaef7ffaef7ffaef7ff
>
```

Our ASCIIHex data is terminated by “>”, the end-of-data marker for the *ASCIIHexDecode* filter; since we are running our image data through the *ASCIIHexDecode* filter on its way to VM, *ReusableStreamDecode* will see this character as end-of-file and cease streaming our data into VM. The *filter* operator will return its filtered-fileobject on the operand stack and the interperter will resume executing the input stream as PostScript code.

```
def
```

This *def* call looks a bit strange standing out all by itself, but remember that this is the first item executed after the *filter* operator returns. When executed, *def* will find on the operand stack the name *ImageData,* which we put there at the start of our program, and the filtered-fileobject returned by the *filter* operator. The *def* operator will tuck those two objects into *userdict* as a key-value pair.

ImageData is now associated with a fileobject that is associated with our image data in VM. We can rewind this virtual file and read its data as often as we like.

Next Page ->

```
/JumpForm <<
...
>> def
```

Now that we have stored our data in memory, we can create our form dictionary. Most of this code is identical to our example from last month, so I won't repeat the detailed description here. What is different (but not *very* different) is the definition of the form's *PaintProc* procedure. There are two lines in this procedure that are different from last month's example:

```
/PaintProc     {
     ...
     ImageData 0 setfileposition
```

This line "rewinds" our virtual file, *imageData,* to the beginning. The *setfileposition* operator takes a file object (our filtered-fileobject, in this case) and an offset from the beginning of the file (*0,* here, indicating the start of the file); it moves the "file pointer" to the specified position in the file. In our case, we reset the file so that the *image* operator, later in *PaintProc,* will read data starting at the beginning of the file.

```
     <<   ...
          /DataSource ImageData
          ...
     >> image
```

In our *PaintProc's* call to *image,* we specify *ImageData* as the source of our image data. Since *ImageData* looks to PostScript like a normal file object, *image* can use it as a data source.

Next Page ->

Acumen Journal: PostScript Tech 16

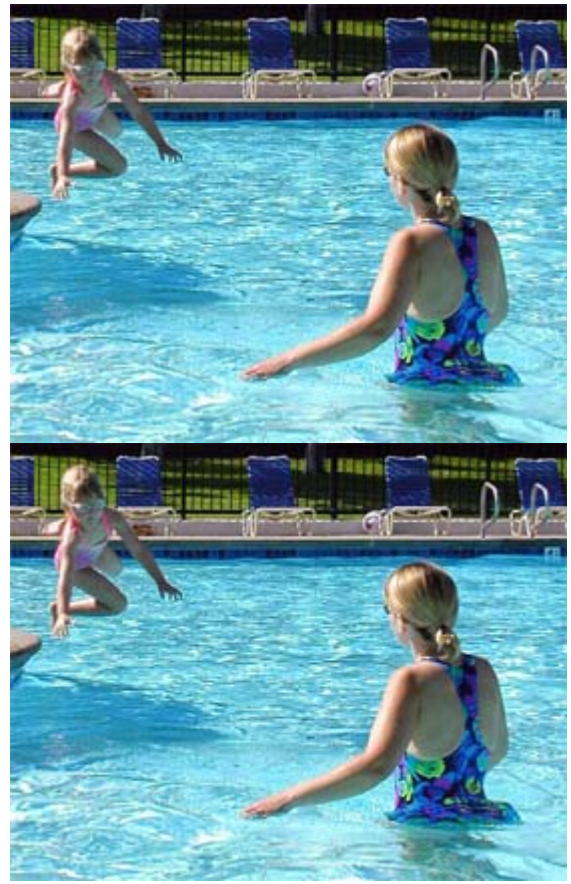
```
...
>> def

JumpForm execform
0 219 translate
JumpForm execform
```

Having defined our form dictionary, we can now execute the form repeatedly. In our case, we execute it twice, producing the output at right. In the first execution, *execform* executes the form's *PaintProc,* drawing the image and caching the painted form. The second call to *execform* draws from the form cache and does not need to execute *PaintProc.*

This accomplishes what we set out to do: we have created a PostScript form that prints an image without the use of a hard disk.

Next Page ->>



Other Data Encodings

Our sample code assumes the image data has been encoded as ASCIIHex. This is very common, but not at all the only possibility. Let's see what we would have to change in our code to accommodate two other common encodings: ASCII85 and raw binary.

ASCII85

The ASCII85 case is easy. Only one thing needs to change in our PostScript code: we will read data into memory through the *ASCII85Decode* filter, rather than *ASCIIHexDecode.* Thus, our call to *filter* looks like this:

This file is in the *ImageForm2.zip* archive as *ReusableStreamA85.ps.*

```
currentfile
<< /Filter /ASCII85Decode >>  % Our data is ASCII85 encoded
/ReusableStreamDecode filter
//^#^%j)u/(aBn8*>B;-'Gqc)*>/r#&eYfk((h<"&ePfe%Lrs^&f2;s*"<8k'c/,)
...
Y=>(e8Y34]T<Vh/^M^^+J'6N"eDZ55ieGtAMUpjJgZXOGjH&-^R,mh/lhI1@qkETY
VXjbdp&D2gs/u+rY4qt!p]%Djs/u+r~>
def
```

Our data now ends with "~>", which is the end-of-data indicator for ASCII85.

The change to ASCII85-encoded data has no affect on the definition or use of our form. That PostScript code is completely unchanged from our original *ReusableStreamDecode* example.

Next Page ->

Acumen Journal: PostScript Tech 18

Raw Binary

Embedding the image data in our PostScript stream as raw binary, rather than in some ASCII-encoded format, is advantageous, since the ASCII encodings substantially increase the size of the data and, therefore, transmission time. *ReusableStreamDecode* has no trouble reading binary data into VM, of course; the trick is telling it when to stop, so that the interpreter can resume treating the input stream as PostScript code. Earlier, we used the end-of-data markers provided by the *ASCIIHexDecode* and *ASCII85Decode* filters to halt *ReusableStreamDecode's* processing of the input stream. There is no equivalent end-of-data intrinsic to the raw data, so how to we tell *ReusableStreamDecode* when to quit?

This file is in the ImageForm2.zip archive as *ReusableStreamRaw.ps.*

We can define our own end-of-data marker if we read the image data through the *SubFileDecode* filter:

```
/ImageData
currentfile
<< /Filter /SubFileDecode                    % Pass data thru SubFileDecode
   /DecodeParms << /EODString (*EOD*) >> % "*EOD*" will be end-of-data
>>
/ReusableStreamDecode filter
,- "□#□□□□□□□□□□□□
... Binary image data goes here ...
t{nmsi□i]=óó.àëqÉ%Åö&É§(àÆ?§Œ)î¿)ò
*EOD*          % This is our end-of-data marker; PS code resumes
def            % Stack: /ImageData fileobj => ---
```

Next Page ->

SubFileDecode is a dummy filter; it has no effect on the data passing through it. What the filter *does* allow you to do is define your own end-of-data indicator. The *DecodeParms* dictionary in the code above (yes, it's "Parms," not "Params"; go figure) has a single key-value pair in it, defining *EODString* to be associated with the string **EOD*.* This string of five characters will be seen by the *SubFileDecode* filter as end-of-data. At this point, *ReusableStreamDecode* will cease reading data, the filter operator will return the filtered-fileobject, and the PostScript interpreter will kick back in.

Once again, sending the image data as binary has absolutely no effect on the remainder of our program. The PostScript code that defines and later executes the form is completely unchanged.

Presuming your communication with the PostScript interpreter transparently passes binary data, this is probably the best way to pass along the image data.

Memory Impact

At first glance, it would seem as though a major advantage to sending our data as raw binary is that it reduces the amount of memory needed to store the data, compared to the ASCII encodings. This actually isn't the case; we passed incoming ASCII data through a decode filter before storing it in memory, so in all three of our cases, we stored the actual binary image data in VM.

Sending the data as raw binary affects only the size of our PostScript stream and, therefore, transmission time.

Next Page ->

Acumen Journal: PostScript Tech 20

ReusableStreamDecode Limitation

Using *ReusableStreamDecode* to supply image data to a form works excellently and is my very favorite way of placing an image into a form. It is, however, subject to one serious restriction: it will work only on LanguageLevel 3 devices. The *ReusableStreamDecode* filter did not exist in Level 2.

If your PostScript code must work on a variety of Level 2 and 3 devices, this is not a good method to use.

Damn.

A Better Alternative

So what do you do if you want to include an image in a PostScript form for a wide variety of printers? They could be Level 2 or Level 3; they may or may not have hard disks.

What you do is pass the image data as an array of strings. Practically speaking, this is the best way of carrying out this task in a world of mixed printers.

Unfortunately, wouldn't you just know it, we're out of space this month.

Next month, then.

Return to Main Menu

Schedule of Classes, Jan – Mar 2003

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 Acrobat class schedule is on the next page.

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	January 27 – 31	March 24 – 28
Advanced PostScript	March 3 – 7	
PostScript for Support Engineers	February 10 – 14	
Jaws Development	On-site only; see the Acumen Training website for more information.	

PostScript Course Fees PostScript classes cost $2,000 per student.

On-Site Classes These classes may also be taught on your organization's site. Go to www.acumentraining.com/onsite.html for more information.

Registration Info →
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 — *No Acrobat classes scheduled for this quarter.* See the Acumen Training website regarding setting up an on-site class.

Interactive Acrobat

Creating Acrobat Forms

Troubleshooting with Enfocus' PitStop

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?

U.K. Classes?

I'm trying to assess how much interest there would be in my conducting occasional (perhaps quarterly) PostScript classes in London as a convenience to European customers. If you would be interested in attending classes in London, please drop me an email. (No commitment, of course; I'm just trying to find out if there is *any* interest in my doing this or if I should be spending my attention elsewhere.)

Creating Acrobat Forms
John Deubert, Adobe Press

"With this book, I wouldn't have been so frustrated. Who knows? Maybe I wouldn't have become the Scourge of Europe!"
— G. Kahn

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 force a public review of your own stomach contents?

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


Standard Security
Specify Password
Password Required to Open Document
User Password:
Password Required to Change Permissions and Passwords
Master Password:
Permissions
Encryption Level:
40-bit RC4 (Acrobat 3.x, 4.x)
No Printing
No Changing the Document
No Content Copying or Extraction, Disable Accessibility
No Adding or Changing Comments and Form Fields
Cancel
OK