Table of Contents

Acumen Training

Embedding Arbitrary Data in Acrobat Files

Relatively few people know that you can embed in a PDF file data taken from any file on your hard disk. The PDF file format includes a mechanism by which arbitrary data—a spreadsheet, a text file, a Word document, a Quark file—can be placed into a PDF file.

There are endless possibilities associated with this ability. An Acrobat expense report form could contain the scanned receipts; a music group's brochure, distributed in PDF file format, could contain a series of mp3 snippets; at an extreme, a PDF document sent out for printing could contain the original QuarkXpress document, fonts, and illustration files from which the document was created.

What makes this PDF characteristic useful is that Acrobat 5 makes it very easy to embed a file in your PDF document and, later, retrieve that file. (Sorry, you can't get to this feature from Acrobat 4.)

Let's see how it all works.

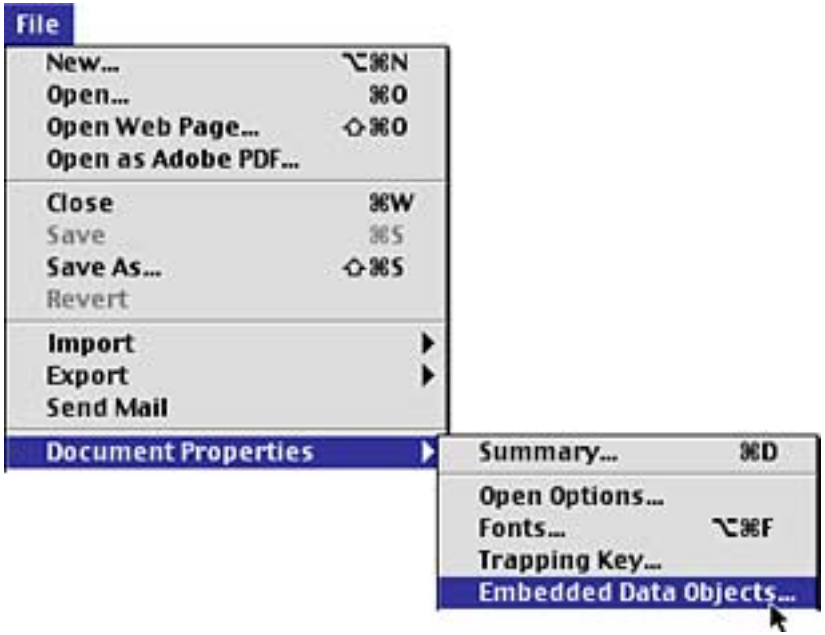
Next page ->

Embedding Data in a PDF File

Embedding data in your PDF document is easy, though access to the feature is located in a somewhat unintuitive place among the Acrobat menus.

With your Acrobat file open, select *File>Document Properties> Embedded Data Objects.* Acrobat will present you with a dialog box that allows you to embed and retrieve data within your PDF file.

Next page ->



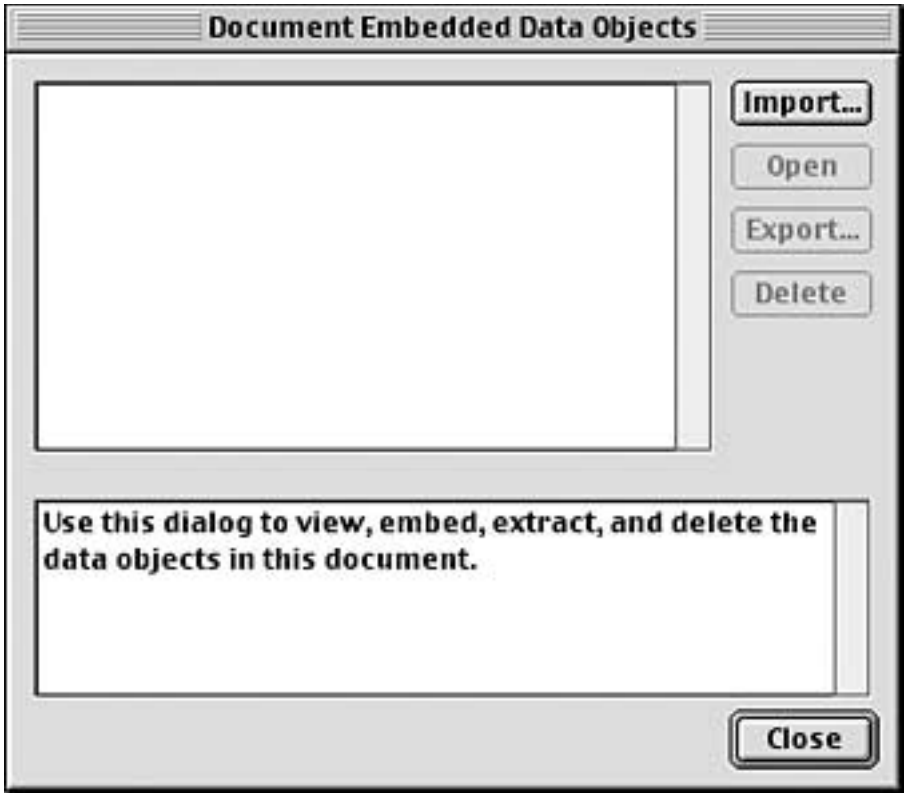
Embedding the Data

The *Embedded Data Objects* dialog box lets you manage the embedding and retrieval of data in your PDF file.

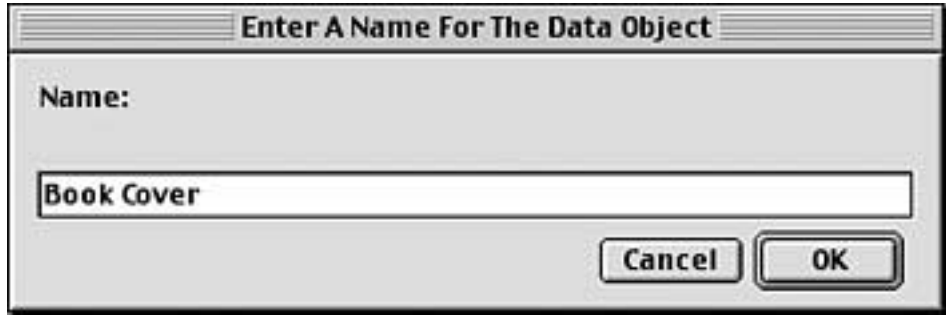
This is every bit as straight-forward as it seems. When you click on the *Import* button, Acrobat presents you with a standard Pick-A-File dialog box, allowing you to choose the file whose contents you want to embed in the PDF document.

Next page ->

Next page ->

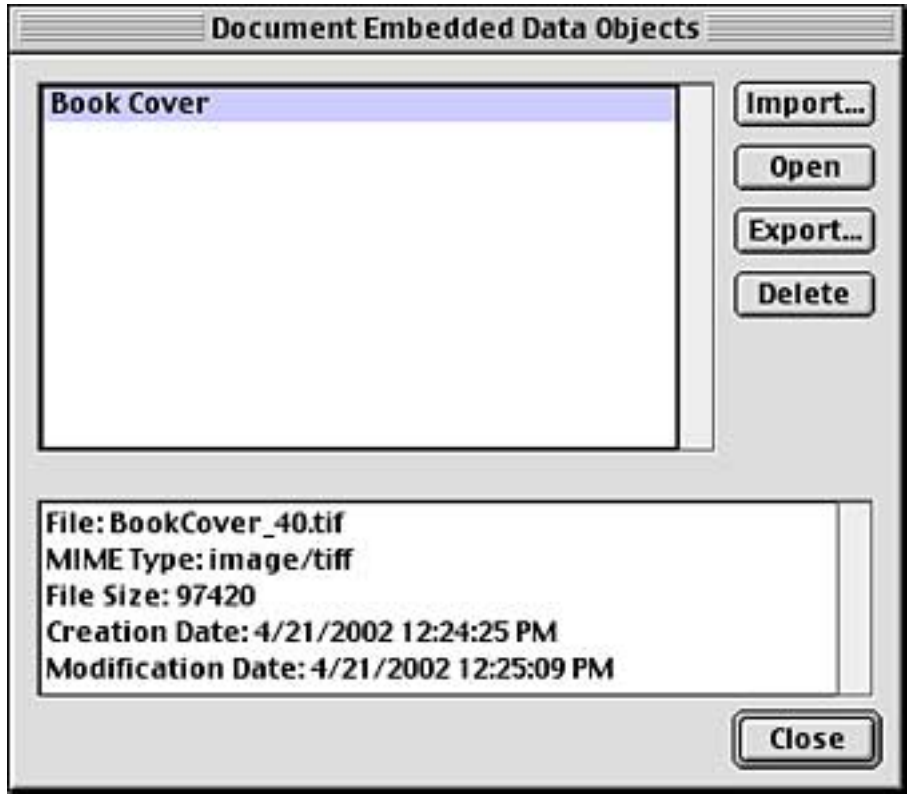


Once you have picked a data file, Acrobat asks you to name the embedded data. This is the name that will appear in the *Embedded Data Object* dialog box.



Note that the name of the data is distinct from the name of the original file. Both names are retained by Acrobat. If you click on a data object in the dialog box, Acrobat will present you with information about the original file.

Next page ->



Retrieving the Data

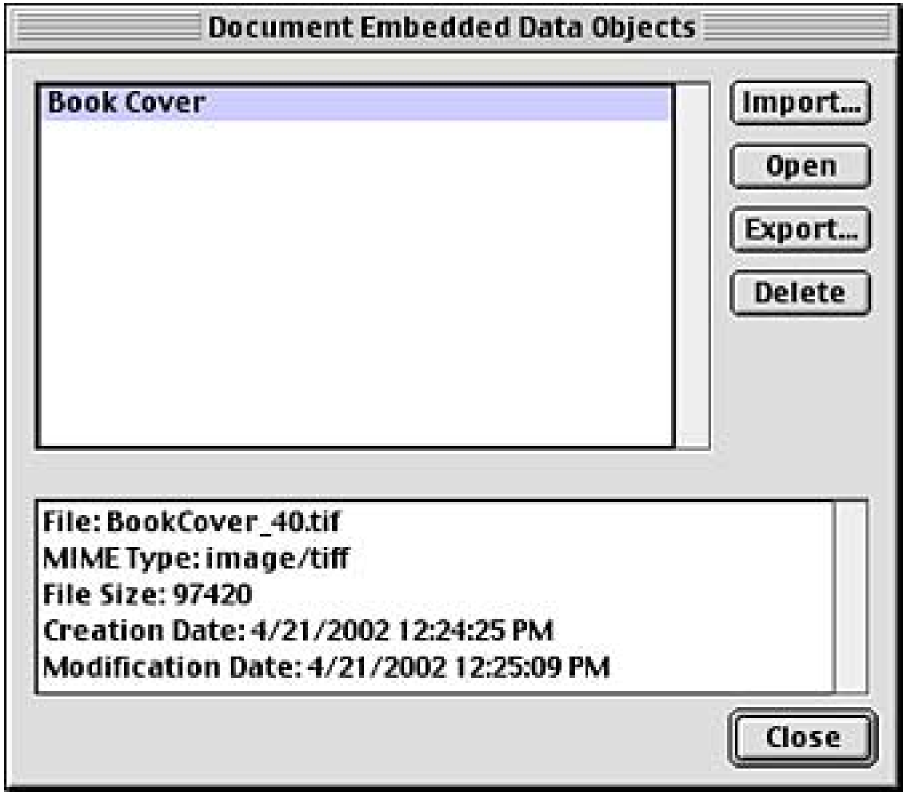
You retrieve embedded data by selecting the data object in the *Embedded Data Objects* dialog box and then clicking on the *Export* button.

Acrobat will give you a chance to specify where the file should be placed on your disk and then retrieve the original file.

Try it out!

I have embedded a very small TIF file in this issue of the *Journal.* Go to *File>Document Properties>Embedded Data Objects* and retrieve it.

Next page ->



Embedding Data in Forms

This discussion assumes that you know how to use the Acrobat form tool to make form fields.

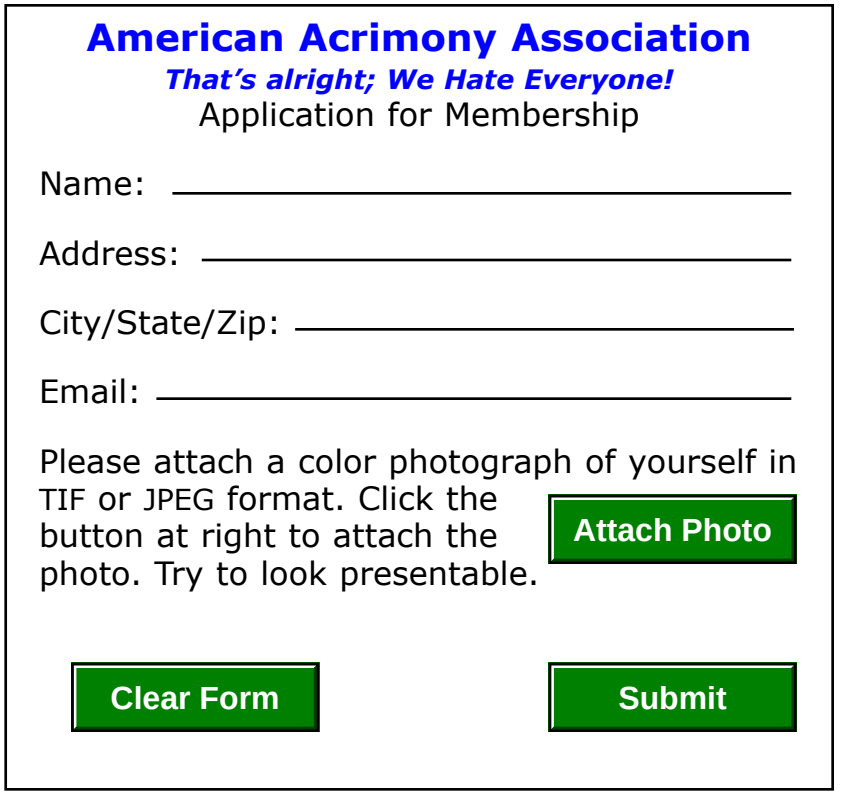
This form is available as a sample file on the Acumen Training training website's Resources page. Look for *Acrimony.pdf.*

One obvious application of this feature in Acrobat is to attach supporting documents to a form.

For example, at right is a membership application form that wants to have attached a photo of the applicant. How can we embed the photo in the form without having to teach the user to access and use the "Embedded Data Objects" dialog box?

The answer is: with a JavaScript.

The "Attach Photo" button will have a JavaScript action attached to its *Mouse Up* event. This JavaScript will make a call to the *importDataObject* method.



The following discussion presumes you know how to make Acrobat forms and have already made the Application form above. (You can download the form, ready made, from www.acumentraining.com/resources.html.)

Next page ->

Attaching the JavaScript

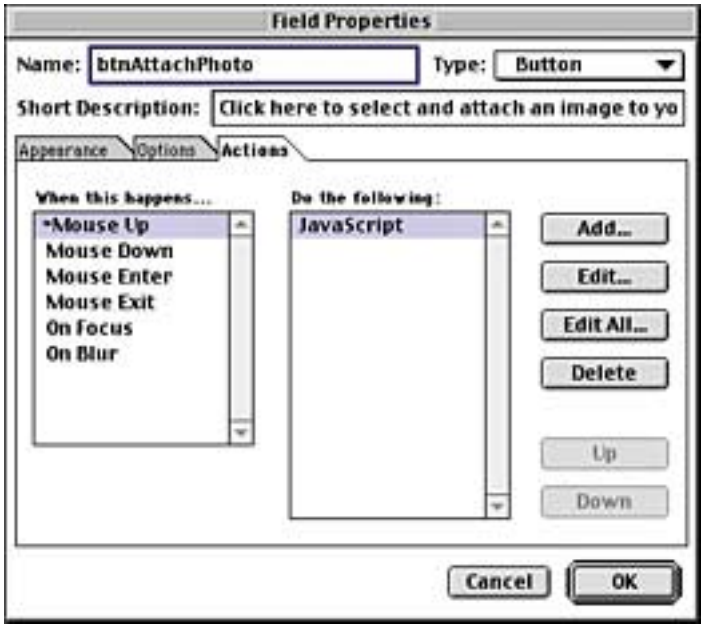
To attach the JavaScript to the "Attach Photo" button:

1. With the Forms Tool selected, double-click on the "Attach Photo" button, gaining access to the button's properties.

2. In the *Actions* panel, click on the *Mouse Up* event and the *Add* button, which takes you to the *Add an Action* dialog box.

3. Select *JavaScript* as the Action Type and click on the *Edit* dialog box.

   At this point, you will be looking at the simple text editor Acrobat provides for typing JavaScripts (next page).



Next page ->

If you don't know how to make buttons and other form fields in Acrobat, you probably should buy and read several copies of *Creating Adobe Acrobat Forms.*

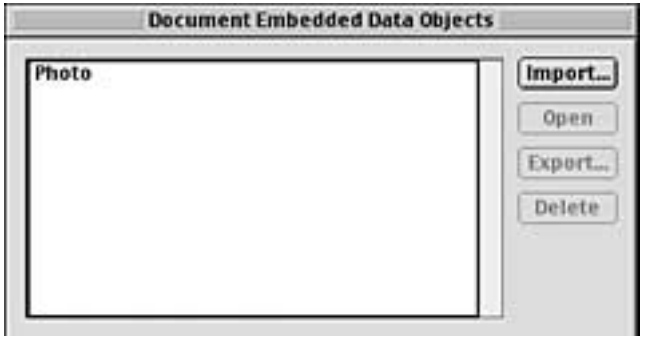
4. Type in the following line of JavaScript:

```
this.importDataObject("Photo");
```

   The text in quotes is the name the data object will have in the PDF file, that is, the name the *Embedded Data Objects* dialog box will display for this data.

5. Repeatedly click *OK* buttons until you return to the form's page.

That's all there is to it. Click on the hand tool and then click on the *Attach Photo* button. Acrobat will ask you to pick a file. When you do so, the contents of that file will be embedded in the form. You can confirm this by opening up the *Embedded Data Objects* dialog box; an item named *Photo* will now appear in the list.



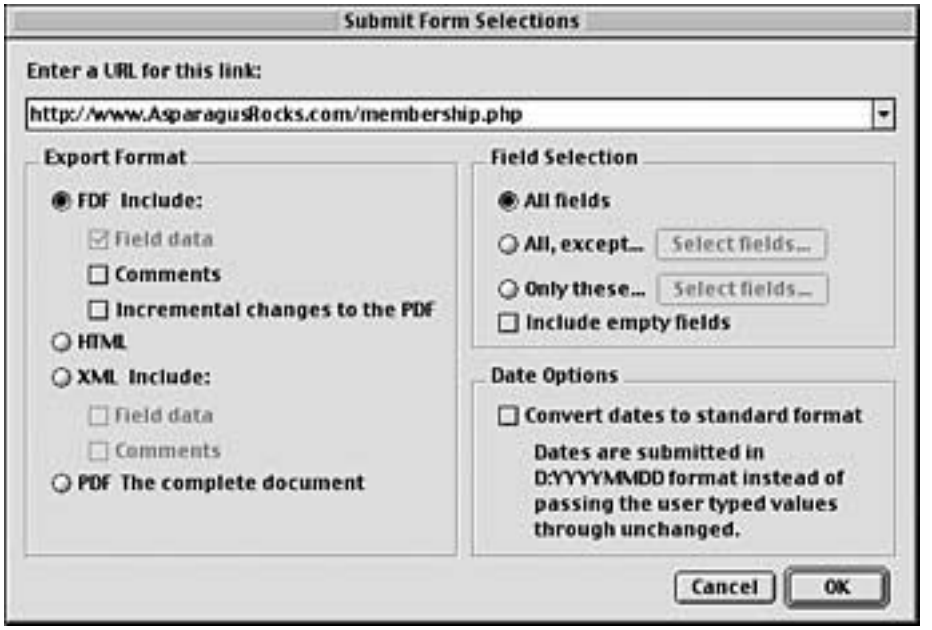
Next page ->

Submitting the Form

A form that contains embeded data must be submitted as FDF (Form Data Format), rather than HTML. The FDF will contain the embedded data, properly tagged and identified.

You could, of course, submit the entire PDF file, instead.

Next page ->



Caveats

A couple of things of which to be aware:

No Type Checking

Acrobat provides no way for checking the nature of the data you embed in an Acrobat file. Our membership application form asks the user to select a TIFF or JPEG file. However, there is no way to ensure that the file the user picks is, in fact, an image file; you cannot prevent the user from mistakenly attaching a Word document.

File Size

Also remember that data embedded in a PDF file will contribute significantly to the size of that PDF file. Don't embed your entire home movie collection into a PDF file and then try to email it to a friend.

Otherwise, there are surprisingly few drawbacks to this technique.

Let me know...

I am curious to know the uses people find or have found for this feature. If you apply data embedding to some interesting problem in form development or PDF file distribution, I'd be very interested in hearing about it. Just drop me a line at john@acumentraining.com.

Return to Main Menu

Isolating PostScript Errors With *SubFileDecode*

Last month, we examined the *SubFileDecode* filter, using it to conditionally ignore unwanted PostScript code without expending time and memory on the unused code.

Let us continue our discussion of *SubFileDecode* with what I think is its most valuable use: isolating the effects of PostScript errors.

Note that I'm going to assume last month's article is still fresh in your mind; you may want to re-read it.

What Problem Are We Solving?

PostScript errors, left to themselves, kill the remaining print job. This is a serious problem for people who do very large print runs. Variable data printing, for example, often entails PostScript print jobs consisting of tens of thousands of concatenated documents, amounting to hundreds of thousands of pages. A PostScript error early in the PostScript stream kills all of the remaining stream; an error in page 27 prevents the printing of pages 28 through 498,060.

If this month's topic sounds familiar, perhaps it's because we discuss it in the *Advanced PostScript* class.

By using SubFileDecode to make each document in the stream into a separate subfile, we can arrange things so that a PostScript error prevents the printing of only the page or document in which the error occurs; the succeeding pages all print correctly.

Let's see how to do it.

Next page ->

Acumen Journal: PostScript Tech

stopped: a Review

First, we need to remind ourselves of the workings of the PostScript *stopped* operator. (We discuss this operator in the *PostScript Foundations* and *PostScript for Support Engineers* classes, so you probably remember this. Don't you?)

The PostScript *stopped* operator takes an executable object—an executable file, procedure body, etc.—from the operand stack and executes it, that is, transfers it to the Execution stack, returning a boolean on the operand stack when the item is finished.

```
execObj   stopped   =>   bool
```

If the interpreter reaches the end of the executable object successfully, without encountering a PostScript error, then *stopped* returns *false.*

On the other hand, if the interpreter encounters a PostScript error while executing the object, execution of the *stopped* object immediately ceases and *stopped* returns *true.*

Thus, *stopped* returns a boolean that will be true if the executable object has any PostScript errors. For example, the following line of PostScript:

```
{ y x div } stopped { (Division failed!!) = } if
```

will attempt to divide *y* by *x* and print the text "Division failed!!" to the output stream if *x* is zero or anything else goes wrong with the division.

Next page ->

Acumen Journal: PostScript Tech 13

stopped and Error Trapping

You can use *stopped* to intercept the default PostScript error handling mechanism, if you wish. Consider the following PostScript code:

```
/ExecuteJob
{    currentfile cvx stopped { (Omigod! An Error!) = } if } bind def

ExecuteJob
72 600 200 100 rectfill
... put more PS code here ...
```

The ExecuteJob procedure, when executed, gets *currentfile,* makes it executable, and then hands the resulting executable file object to *stopped.* The entire rest of the PostScript stream is being executed in our *stopped* context.

If there are no PostScript errors in the code following the invocation of *ExecJob*, *stopped* will return *false* when we reach the end of the PostScript stream. If there is a PostScript error in the PostScript stream, the interpreter will immediately remove from the Execution stack *ExecuteJob's* pointer to currentfile and *stopped* will return true; we then print an error message ("Omigod! An Error!") to *stdout.*

Next page ->

Acumen Journal: PostScript Tech 14

SubFileDecode and Errors

Returning to our main topic, in order to prevent PostScript errors from poisoning the entire print job, we shall attach the *SubFileDecode* filter to *currentfile* and then execute the combination with *stopped:*

All of the examples in this article are available on the Acumen Training Resources page. Look for *SubFileDecode.zip.*

The example at right is stored within the zip file as *SubFileDecode 0.ps*.

```
% Attach SubFileDecode filter to currentfile; name it "psSrc"
/psSrc currentfile 0 (*END*) /SubFileDecode filter def
psSrc cvx stopped             % Execute currentfile through the filter

72 600 moveto
(Page 1) show                 % Error! No current font.
showpage
*END*                         % End of subfile;

/psSrc currentfile 0 (*END*) /SubFileDecode filter def % Do it again
psSrc cvx stopped                                      % for page 2

/Helvetica 20 selectfont
72 600 moveto
(Page 2) show
showpage
*END*
```

There is a PostScript error in page 1 of this two-page PostScript document; nonetheless, page 2 will still print. We have made each page a separate subfile; the error in Page 1 prevents the printing of that subfile, but not the following ones.

Next page ->

Acumen Journal: PostScript Tech 15

Step by Step Let's look at this in more detail.

```
/psSrc currentfile 0 (*END*) /SubFileDecode filter def
```

In this first line, we attach the *SubFileDecode* filter to *currentfile.* The text "*END*" will mark the end of the subfile and we shall skip zero instances of this text. (That is, the first instance of "*END*" in the stream will mark the end of the subfile.)

We give the filtered file object the name "psSrc."

```
psSrc cvx stopped
```

Convert *psSrc* (currentfile + SubFileDecode) to executable and execute it with *stopped.*

```
72 600 moveto
(Page 1) show
showpage
```

This PostScript code is executed through the *SubFileDecode* filter. When the interpreter encounters the *invalidfont* error provoked by the *show, psSrc* will be popped off the Execution Stack and *stopped* will return a *true* on the operand stack.

Execution will then procede with whatever is left in *currentfile.* In our case, this will be the second page's Postscript code.

```
*END*
```

This marks the end of the first page's subfile.

Next page ->

Acumen Journal: PostScript Tech 16

Error Reporting

In our previous example, the error is handled silently: no error message is reported, the page on which the error occurs is simply not printed. It would be much more useful to implement some sort of error reporting.

To do so, we must make use of the Boolean value that *stopped* leaves on the stack, something that we ignored in our previous example.

This program is named *SubFileDecode 1.ps.*

```
/_ProgressDict_      <<    % Holds "state" info (currently just page #)
     /PageNumber 0
>> def

/HandleError              % This reports error info; currently just page #
{    (Error in page )print
     _ProgressDict_ /PageNumber get =
} def

_ProgressDict_ /PageNumber 1 put
/strm currentfile 0 (*END*) /SubFileDecode filter def
strm cvx stopped

72 600 moveto (Page 1) show showpage

*END*                     % stopped leaves a Boolean on the op stack
{ HandleError } if        % report error if the bool is true
% ... more pages of PS ...
```

Next page ->

Acumen Journal: PostScript Tech 17

Step-by-Step

```
/_ProgressDict_     <<
     /PageNumber 0
>> def
```

Here we define a dictionary that keeps track of state information for error reporting. In our example, we are keeping track of only the current page number; in a real situation, you might want to also store such things as document title (if the PostScript stream prints multiple documents), client name, or whatever else would identify the specific location within a long print stream.

We could simply do a *def,* putting this information into *userdict,* but in the long run here are benefits to putting it into a dictionary specific to the purpose.

```
/HandleError
{    (Error in page )print
     _ProgressDict_ /PageNumber get =
} def
```

Our error handling procedure prints the information kept in *_ProgressDict_.* It does not (yet) print the error name or other information about the error.

```
_ProgressDict_ /PageNumber 1 put
```

We set up the state information in *_ProgressDict_* to reflect the upcoming subfile. Here, we simply set the value of *PageNumber;* in real life, this would probably be quite an extensive collection of information to be set.

Next page ->

```
/strm currentfile 0 (*END*) /SubFileDecode filter def
strm cvx stopped

72 600 moveto (Page 1) show showpage
```

We create and execute our subfile, as before.

```
*END*
```

This is the end of our subfile. At this point in our execution, the *stopped* that executed the subfile will place a boolean on the stack. A *true* will indicate that a PostScript error has taken place.

Warning: I'm glossing over a very important under-the-hood detail here. We'll discuss it in a moment.

```
{ HandleError } if
```

Finally, we execute *HandleError* if the boolean returned by *stopped* is *true.*

So, Where Are We?

At this point, we have constructed a template for a PostScript stream consisting of a series of logical subfiles. A PostScript error in one subfile will not prevent the printing of the subfiles that follow. Our sample code can be readily generalized and applied to a very wide variety of situations.

The is only one tiny problem: the error handling as we have implemented it in our example won't work in the general case. The problem is subtle, but fixable.

Next page ->

Acumen Journal: PostScript Tech 19

Error Handling, Second Pass

Consider the execution of the subfile in our previous example:

```
/strm currentfile 0 (*END*) /SubFileDecode filter def
strm cvx stopped

...PostScript Code...

*END*
{ HandleError } if
```

stopped, Under the Hood

When there is an error in the subfile's PostScript code, how does *stopped* know to skip to the end of the subfile before putting its boolean on the stack? The answer is: it doesn't.

The *SubFileDecode* filter—any filter, for that matter—maintains its own input buffer; when it needs data, it reads it from *currentfile,* streaming the PostScript code into its buffer until either the buffer is full or it hits end of file. In our case, because our subfile has so little PostScript in it, our entire subfile fits in the *SubFileDecode* buffer; the filter reads to the end-of-subfile on the first try.

When a PostScript error is encountered, *stopped* ceases executing our *currentfile*-plus-*SubFileDecode* combination. Whatever PostScript remains in *SubFileDecode's* input buffer is discarded and the interpreter resumes executing *currentfile* directly, beginning at whatever code followed the last bufferful read by *SubFileDecode.*

Next page ->

Acumen Journal: PostScript Tech 20

Since, in our example, the subfile fits entirely within the filter's buffer, *SubFileDecode* reads and buffers everything up through the end-of-subfile, **END*,* when *stopped* returns, the interpreter picks up execution with our conditional execution of *HandleError.*

```
{ HandleError } if
```

The Problem

What if we have a large amount of PostScript code in our subfile, so that the entire subfile does not fit in *SubFileDecode's* input buffer? If a PostScript error occurs early in the subfile, the interpreter will resume directly executing *currentfile* at whatever point in the PostScript stream happens to follow the current *SubFileDecode* buffer; this will be a functionally random place in the subfile's PostScript code and will almost certainly generate an error of some flavor.

So, What Do We Do?

To avoid this problem, we need to "manually" advance *currentfile* to the end of the subfile before we let the interpreter resume executing *currentfile.* To do this, we need to rearrange our Postscript code a bit.

Next page ->

Acumen Journal: PostScript Tech 21

The New Code…

This program is named *SubFileDecode 2.ps.*

```
/_ProgressDict_      <<
     /PageNumber 1
>> def

/BeginSubFile        % pageNum => ---
{    % Save the current page number into _ProgressDict_
     _ProgressDict_ /PageNumber 3 -1 roll put
     % Attach SubFileDecode to currentfile; name it strm
     /strm currentfile 0 (*END*) /SubFileDecode filter def
     % Execute the filtered stream and handle the return value
     strm cvx stopped { strm flushfile HandleError } if
} bind def

/HandleError         % --- => ---
{    (Error in page )print _ProgressDict_ /PageNumber get =
     errordict /handleerror get exec    % Execute the default error hdlr
} bind def

1 BeginSubFile
...PS Code...
*END*

2 BeginSubFile
...PS Code...
*END*
```

Next page ->

Acumen Journal: PostScript Tech 22

Step By Step

```
/BeginSubFile        % pageNum => ---
{    _ProgressDict_ /PageNumber 3 -1 roll put
     /strm currentfile 0 (*EOF*) /SubFileDecode filter def
     strm cvx stopped { strm flushfile HandleError } if
} bind def
```

The *BeginSubFile* procedure provides our problem's solution. This procedure creates and executes our subfile and, most importantly, handles the boolean returned by *stopped.*

When *stopped* returns, the interpreter resumes execution of the *BeginSubFile* procedure. If the subfile contains a PostScript error, *stopped* returns true and our *if* procedure will flush the remaining subfile and then execute *HandleError.*

Note that, as a convenience, *BeginSubFile* takes the current page number as an argument and places it into *_ProgressDict_.*

```
/HandleError         % --- => ---
{    (Error in page )print _ProgressDict_ /PageNumber get =
     errordict /handleerror get exec
} bind def
```

Our *HandleError* procedure, as before, prints the page number on which the error occurred. We've added something new, however: *HandleError* now goes to *errordict* and executes the default PostScript error reporting procedure, *handleerror.* This allows us to see the standard PostScript error message, including the error name and offending command.

Next page ->

```
1 BeginSubFile
...
... PS Code
...
*END*

2 BeginSubFile
...
... PS Code
...
*END*
```

Our template now consists of an invocation of *BeginSubFile* at the beginning of each subfile and **END** at the end of each subfile.

Caveats (none, really)

This technique for isolating the effect of PostScript errors is without any disadvantages that I've encoutered or heard of. It is, on the other hand, vastly important for people who have very long print runs or for people who routinely concatenat PostScript files from a variety of sources and then send them to a PostScript printer.

Return to Main Menu

Schedule of Classes, July – Sept, 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 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	July 29 – Aug 2
Advanced PostScript	August 12 – 16
PostScript for Support Engineers	Sept 30 – Oct 4
Jaws Development	

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	Sept 5 (½-day, morning)
Interactive Acrobat	
Creating Acrobat Forms	Sept 5 (½-day, afternoon)
Troubleshooting with Enfocus' PitStop	

Acrobat Class Fees

Acrobat Essentials and *Creating Acrobat Forms* (½-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?

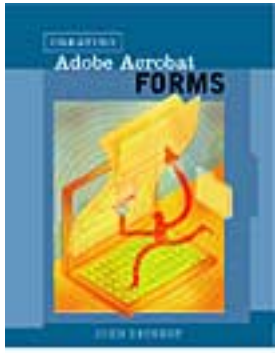
John Does Contracting

Don't let's forget that John also does contract work in PostScript and Acrobat. If you have a need for PostScript programming or Acrobat form development, drop me an email or give me a call.

It's hard to find anyone with more experience and knowledge in the field.

Creating Acrobat Forms

Have you bought your copy yet?



And why ever not?

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