Table of Contents

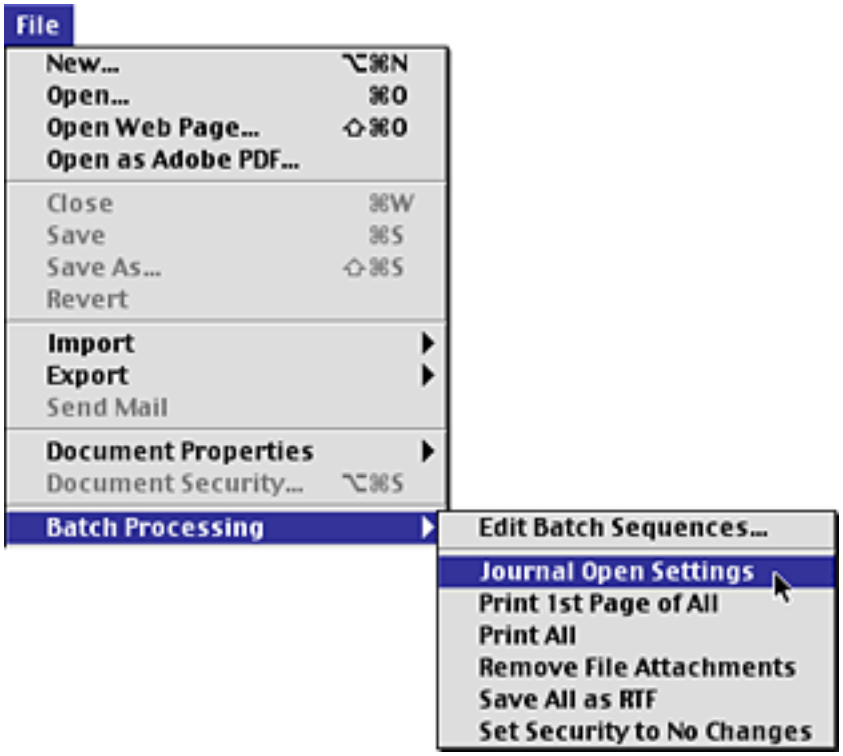
Acumen Training

Acrobat Batch Processing

Many people are unaware that Acrobat has quite a useful batch processing mechanism that lets you perform automated actions upon one or more PDF files. Simply select a sequence from the *File>Batch* submenu and Acrobat will carry out the actions that make up that sequence.

Depending upon the sequence, the actions may be applied to the current document, a folder full of PDF files, or a file selected when you execute the sequence.

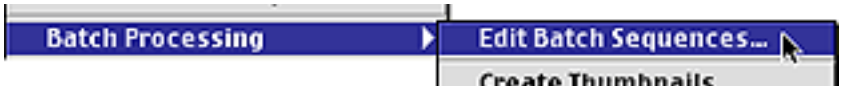
This month, we shall see how to create and use our own batch sequences.

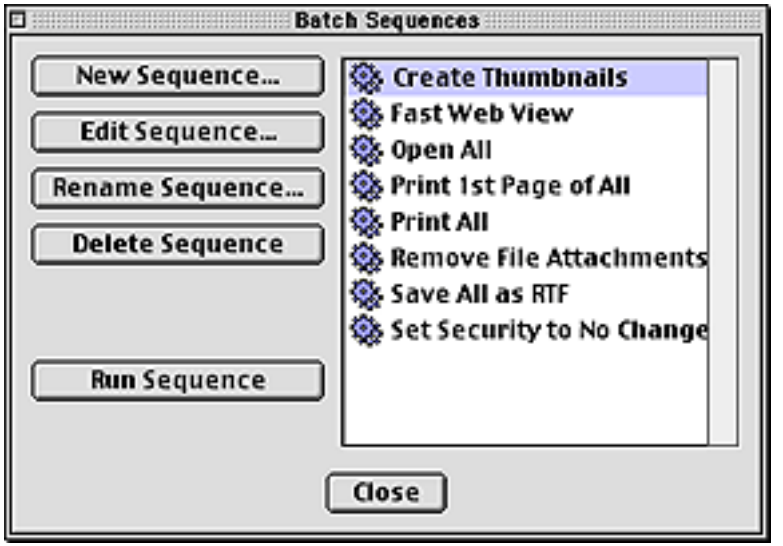


Next Page ->

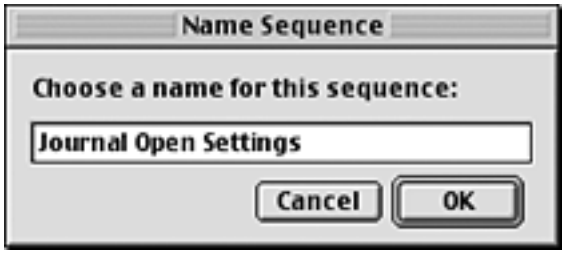
Creating a Sequence

If you select *File>Batch Processing> Edit Batch Sequences…,* Acrobat will present you with a dialog box that lets you rename, delete, and otherwise manage the currently-defined sequences. Of interest to us here is the *New Sequence…* button.





When you click on this button, Acrobat asks you for a name for then new sequence. You can type in any short, descriptive name for your sequence.

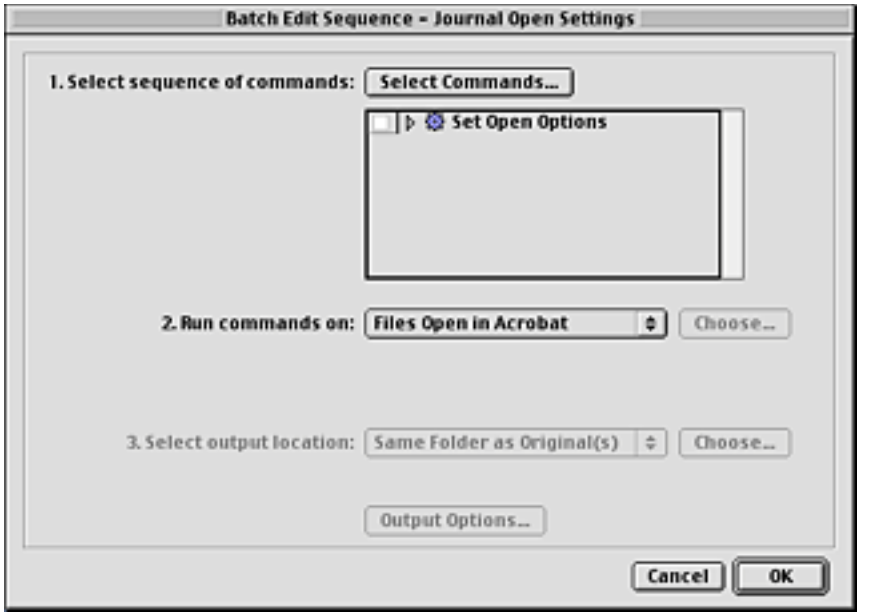


Click the *OK* button and Acrobat presents you with the *Batch Edit Sequence* dialog box (next page) that we use to specify the characteristics of our new sequence.

Next Page ->

Setting Up the Sequence

The *Batch Edit Sequence* dialog box lets you specify the three properties that characterize a batch sequence:



1. The commands that make up the sequence.

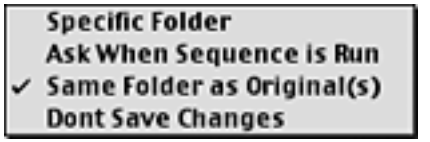
   Click the *Select Commands…* button to specify what should actually happen when this sequence executes.

2. To what files the sequence should apply.

   You can select from among four choices in a pop-up menu, as at right. If you select *Selected Files* or *Selected Folder,* you will need to click the *Choose* button to specify the files or folder to which the sequence should apply.

3. The *output location,* that is, the location into which a PDF file should be saved after being modified by our sequence.

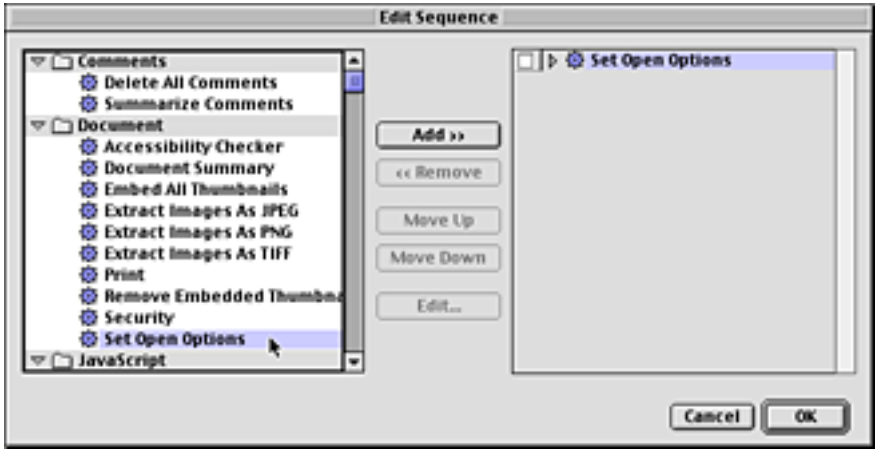
   The *Output Options* button lets you specify the name of the file to which the modified document should be saved. I'll let you explore this on your own.

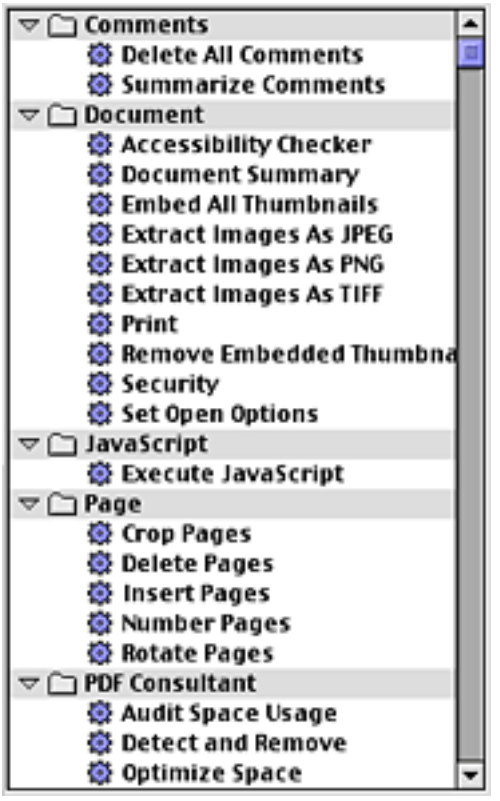


Next Page ->

Specifying the Sequence

Clicking the *Select Commands* button in the *Batch Edit Sequence* dialog box presents you with the *Edit Sequence* dialog box. This lets you specify the actions that will make up your sequence.





You choose these actions from a predefined list on the left side of the dialog box. To add one of these actions to your sequence, select that action in the list and click on the *Add* button. You may add as many of these actions to your sequence as you wish.

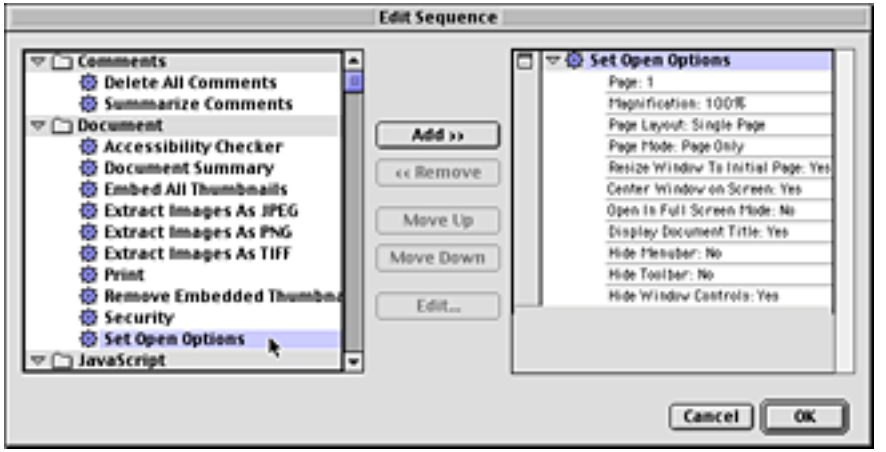
For a complete discussion of the Acrobat *Document Open* options, see the December 2001 *Acumen Journal.*

For the purpose of discussion, I'll assume that we are adding *Set Open Options* to our sequence, as in the illustration above. This action sets the *Document Open* properties of a PDF file (the initial window size, magnification, etc.)

Next Page ->>

Action Settings

When you click on the disclosure button (the little "twisty triangle" on the Mac) for an action in your sequence, Acrobat displays the settings associated with that action, as at right. For example, in the case of a *Set Open Options* action, Acrobat will show you the *Document Open* options set by the action.



Changing the Settings

To change the settings associated with an action in your sequence, double-click that action in the right-hand list. Acrobat will present you with a dialog box that lets you specify the details of that action. This dialog box will be different for each type of action.

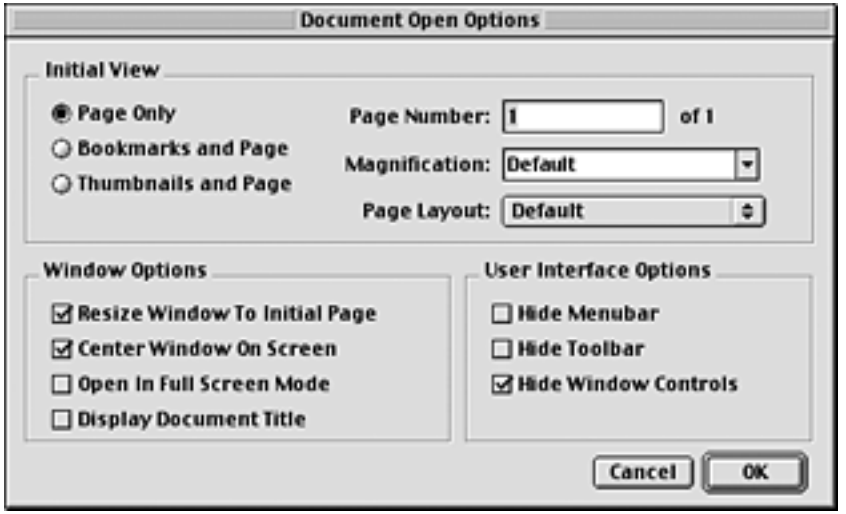
At right is the dialog box for the *Set Open Options* action. This presents all the *Document Open* properties available in Acrobat. In this dialog box, you specify how this action should set these properties.

Note you can set a property to *Leave As Is,* indicating it should not be modified by the sequence.

Next Page ->

"Interactive" Switch

There is a little icon to the left of each item in your sequence's list of actions. Clicking this icon toggles "interactive mode" for that action.



If interactive mode is turned on, the action will carry out the activity you specified and then present you with the regular Acrobat dialog box that performs that same action. (In the case of the *Set Open Options* action, this would be the standard *Document Open Options* dialog box, at right.)

Interactive mode lets you make changes to the settings each time you run the sequence. (In our *Open Options,* perhaps we always want to open to page one, but we want to set the default magnification on a case-by-case basis.)

Done!

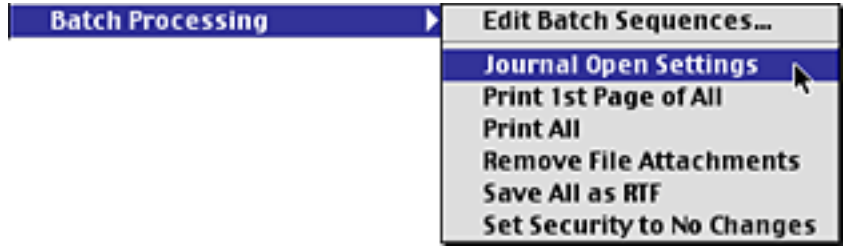
Once you have specified all of the actions that belong in your sequence, and the settings for each action, you can exit out of all the dialog boxes. Your sequence is now ready to use.

Next Page ->

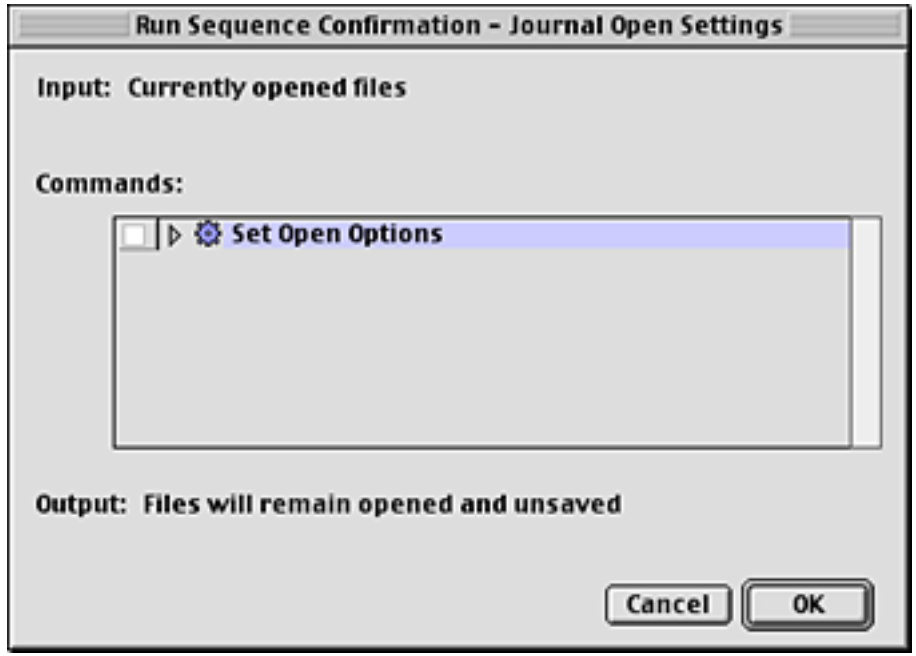
Acumen Journal: Acrobat User 7

Using Your Sequence

Once you have created a sequence, it will appear in the Acrobat *Batch Processing* submenu. To execute your new sequence, simply select it in the submenu.



By default, before executing a sequence, Acrobat will ask if you're sure you want to do so; you can turn off this confirmation request in Acrobat's preferences.



I have found Acrobat's batch sequences to be quite useful and remarkably easy to use. They're not nearly as fully featured as, say, the Action List mechanism in Enfocus' *PitStop,* but are certainly something I find myself using frequently.

Return to Main Menu

Using Images in Forms, Part 3

This article assumes you have read the previous two articles. If you haven't done so, do so. You can get them from the *Acumen Journal* web page.

For the past two months, we have looked at ways to incorporate scanned images into a PostScript form. The difficulty associated with this task is supplying image data to our form's *PaintProc* procedure; there is no way of predicting when this procedure will be called by the form mechanism.

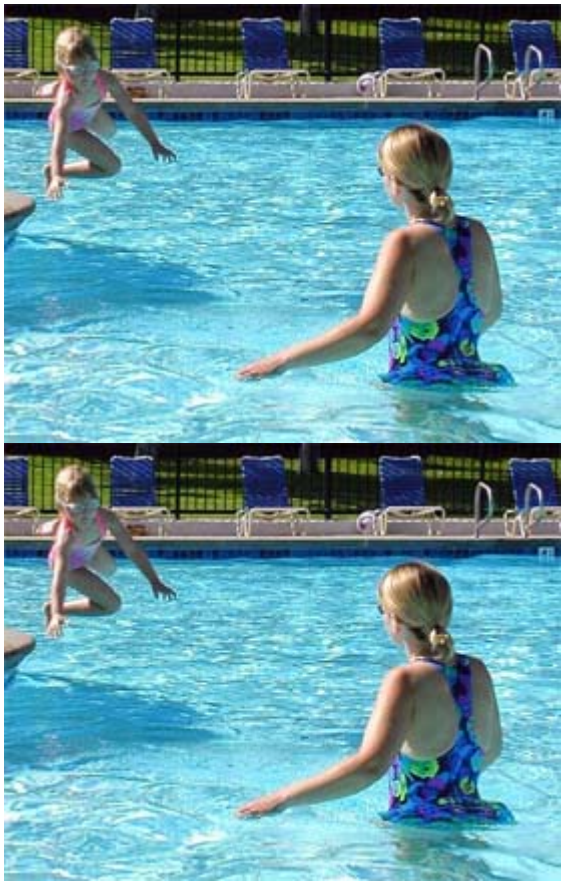
So far, we have examined two methods for providing data to *PaintProc,* each with its virtues and problems:

- Save the image data to disk.

  This works very well, but requires the PostScript interpreter have a disk available to it; this is very frequently not the case.

- Save the image data into memory using the *ReusableStreamDecode* filter.

  This, too, works very well, but will work only on PostScript 3 devices, since *ReusableStreamDecode* did not exist in Level 1 or 2.



This month, we shall conclude the series with a look at a third method that lacks the disadvantages of the other two methods: storing the data in an array of strings.

Next Page ->

An Overview

What we're going to do is put our image data into an array of strings:

```
/imagedata [
   (...first dollop of data...)
   (...second dollop of data...)
   ...
   (...nth dollop of data...)
] def
```

We discuss data acquisition procedures in the *PostScript Foundations* and *Support Engineers* classes. Go back and check your notes for a review.

Our call to the *image* operator will then use a *data acquisition procedure* (DAP, for short) for its data source. Remember data acquisition procedures? When used as a source of image data, it is called by the *image* operator whenever it needs more data; the procedure is expected to return a string containing image data.

In our case, every time *image* calls our DAP, the procedure will simply fetch the next data string from our array and leave it on the stack as its return value:

```
{ imagedata i get
  /i i 1 add store
}
```

Note that we shall need to keep an externally-defined variable as our index into the array.

The nice thing about this approach is that it will work on any LanguageLevel 2 or 3 printer; no hard disk is needed.

Next Page ->

Acumen Journal: PostScript Tech 10

Constructing the Data Array

Before we dive into the PostScript code, let's address ourselves to a preliminary issue: how do we create this array of image data strings? I don't much want to do what is on the previous page: have a series of parentheses, each containing many kilobytes of image data:

```
/imagedata [
   (...first dollop of data...)
   ...
   (...nth dollop of data...)
] def
```

Among other things, an occasional byte of image data will happen to be the ASCII code for a parenthesis or a backslash; since these have meaning to the PostScript scanner when it's constructing a string, they will poison the creation of that string. We could preprocess the data, preceding "special" characters with a backslash, but I'd very seriously rather not.

It would be nice if we could come up with a more convenient way of constructing the data array, preferably something that would let us just dump image data into the input stream.

I wouldn't bring this up if we couldn't do it, of course.

As we shall see, our PostScript code will start by defining a procedure, *CreateDataArray,* that reads data from a file *(currentfile,* in our case) and places it into an array of strings.

Next Page ->

Acumen Journal: PostScript Tech 11

The PostScript

This sample program mirrors the previous months' examples. It places an image ("The Jumping Granddaughter") into a form and then uses the form to print the image twice, as on the first page of this article.

Create the String Array

```
/CreateDataArray      %  srcfileobj => [(Array)(of)(Strings)...]
{
     /temp exch def   % Save the file object
     [                % Begin our array (puts a mark on the stack)
     {                % Begin our "loop" loop
       temp 16384 string readstring     % Read data into a new string
       not { exit } if                  % Exit if we are out of data
     } loop                             % Otherwise, go back & do it again
     ]                % Create the array
} bind def

% Create an array of strings, reading data from currentfile
/ImageData
currentfile /ASCIIHexDecode filter CreateDataArray
2c192d200f1f2213182319181d14171914181b161d121117121212141611
16191211140d0e0e0e13121716131c0d0b161517240d111c12151c13171a
...
0f0e131414161d181e1611181f17241a1221170c1d160b1b190e1e1c121d
>
def
```

This sample program is in the file *ImageForm3.zip* on the Acumen Training *Resources* page.

Next Page ->

Acumen Journal: PostScript Tech 12

Define the Form

```
/JumpForm <<                              % This is the same form as in the
     /FormType 1                          % previous two Journal articles.
     /BBox [ 0 0 278 219 ]                % The changes are commented below.
     /Matrix [ 1 0 0 1 0 0 ]

     /PaintProc
     {    pop
          userdict /i 0 put                    % Initialize our index
          /DeviceRGB setcolorspace
          278 219 scale
          <<   /ImageType 1
               /Width 278
               /Height 219
               /BitsPerComponent 8
               /ImageMatrix [ 278 0 0 -219 0 219 ]
               /DataSource    {               % Here's our Data Acq. Proc.
                    ImageData i get          % Get next string in the array...
                    /i i 1 add store         % ...and increment i in userdict
               } bind
               /Decode [ 0 1 0 1 0 1 ]
          >> image
     } bind
>> def
```

Use the Form

```
JumpForm execform
0 219 translate
JumpForm execform
```

Next Page ->

Acumen Journal: PostScript Tech 13

Stepping Thru' the Code

Defining the Procedure

Our program starts by defining a procedure that creates an array of strings from data read from a fileobj, *currentfile,* in our case.

```
/CreateDataArray     %     srcfileobj => [(Array)(of)(Strings)...]
{
```

The *CreateDataArray* procedure takes a fileobject as its argument. It reads to the end of that file, storing the data it receives into an array of 16k strings, returning that array on the stack.

You will invariably want to attach a filter to your source fileobject so that you can tell the procedure where the end of the data is. Otherwise, the procedure will happily read the entire rest of your PostScript program into the array, including whatever executable PostScript code follows the call to *CreateDataArray.* We'll come back to this topic in a moment.

```
/temp exch def    % Save the file object
```

Our procedure starts by saving the fileobject into a variable, *temp.* I admit to an aesthetic bias against saving procedure arguments into variables; it seems so un-PostScript. It sometimes happens, however, that it is much clunkier to juggle repeatedly-used stack elements than to just put the arguments into variables; such is the case here.

Next Page ->

Acumen Journal: PostScript Tech 14

```
[                              % Begin our array (puts a mark on the stack)
```

Here we begin our array. Remember (from your PostScript classes) that the open bracket merely puts a *mark* object on the stack. The array will actually be created later, by the close bracket operator.

```
{                                              % Begin our "loop" loop
     temp 16384 string readstring              % Read data into a new string
```

We now start an indefinite *loop* loop that reads incoming data one 16k buffer at a time, leaving each string of data on the stack. This loop exit upon end-of-file.

The first line in our loop reads data from our source file into a newly-made, 16-kilobyte string. The *readstring* operator leaves this string, now full of data, on the operand stack; it also returns a Boolean object that will be *false* if we are at end-of-file.

The choice of string size is dictated by two facts:

- A large string size will require fewer strings to hold the data. Remember that our loop exits with *all* of the data strings on the operand stack; reading a 1 megabyte image into 16k strings would leave 67 strings on the stack. If the strings are too small and the data is too large, you could provoke a *stackoverflow* error.

- Small strings will minimize wasted VM in the final call to *readstring.* That final call will probably return a partially filled string. The string's *length* attribute will be set to the amount of data actually read, but the memory allocated for the string will still be 16k (or whatever). The smaller your strings, the less the possible memory waste.

I chose a 16k string size as a balance between these two requirements.

Next Page ->

```
        not { exit } if                 % Exit if we are out of data
    } loop
```

The Boolean returned by *readstring* will be false at end-of-file. We reverse this Boolean with the *not* operator and then exit from our loop if the reversed Boolean is true.

This completes our loop, which will repeatedly execute until it reaches the end of the source file. At this point, the loop exits with all of the data strings piled up on the stack.

```
    ]
    } def
```

Our procedure ends by finishing the array construction with a close bracket. This collects everything off the stack down through the mark, creates an array containing the former stack contents, and leaves that array on the stack. This array is the return value of our procedure.

Creating the String Array

```
/ImageData
```

We next want to use our newly-defined procedure to create a named array of strings from our image data. We start by pushing the name *ImageData* onto the operand stack. This will eventually be the name of our array; for the moment, it's just a name object sitting on the stack.

Next Page ->

```
currentfile /ASCIIHexDecode filter CreateDataArray
2c192d200f1f2213182319181d14171914181b161d121117121212141611
16191211140d0e0e0e13121716131c0d0b161517240d111c12151c13171a
...
a1edfba2f0fda6f2ffa9f5ffaef6ffaef7ffaef7ffaef7ffaef7ffaef7ff
>
```

Here is where we actually make the data array. Our data source is *currentfile* to which we are attaching the *ASCIIHexDecode* filter. This will convert our incoming ASCII data into the original binary, which gets stored into the strings. (We don't want to store the data in ASCII form because that would double the amount of VM it occupies.)

The *ASCIIHexDecode* filter also lets us identify the end of the data; the ">" symbol at the end of the image data is the filter's end-of-data marker; this will be seen by *ASCIIHexDecode* as logical end-of-file. This is where *CreateDataArray* will stop reading data and where the PostScript interpreter will resume executing our PostScript code.

```
def
```

When *CreateDataArray* ceases execution, there will be two items left on the stack: the array of strings created by the procedure and, beneath that, the name *ImageData,* which we placed on the stack before executing *CreateDataArray.* Our call to *def* places the name and the array into the current dictionary as a key-value pair.

Next Page ->

Acumen Journal: PostScript Tech 17

Defining the Form

```
JumpForm  <<
     /FormType 1
     ...
>> def
```

Our form definition is almost exactly identical to those in the previous *Journal* articles. If you haven't done so, go back and reread the December 2002 article, which describes the workings of this form in detail. Here, I'm going to look only at the few lines that have changed in the form's *PaintProc* procedure.

```
/PaintProc     {
     pop
     userdict /i 0 put        % Initialize our index
```

Our *PaintProc* procedure, after throwing away its copy of the form dictionary (read December 2002!) defines a variable, *i,* having a value of *0.* This will be our index into the string array.

We are breaking a philosophical principal, here, by the way. PostScript forms should be self-contained; *PaintProc* should make references only to items defined in *systemdict* or in the form dictionary, itself. We are defining *i* into userdict, which is bad form (so to speak).

Unfortunately, we cannot define *i* into the form dictionary, itself, since that dictionary is read-only; *PaintProc* wouldn't be able to create the variable, let alone increment it, if *i* were defined into the form dictionary. That being the case, we shall simply define *i* into *userdict* and hope no one sneers at us too openly.

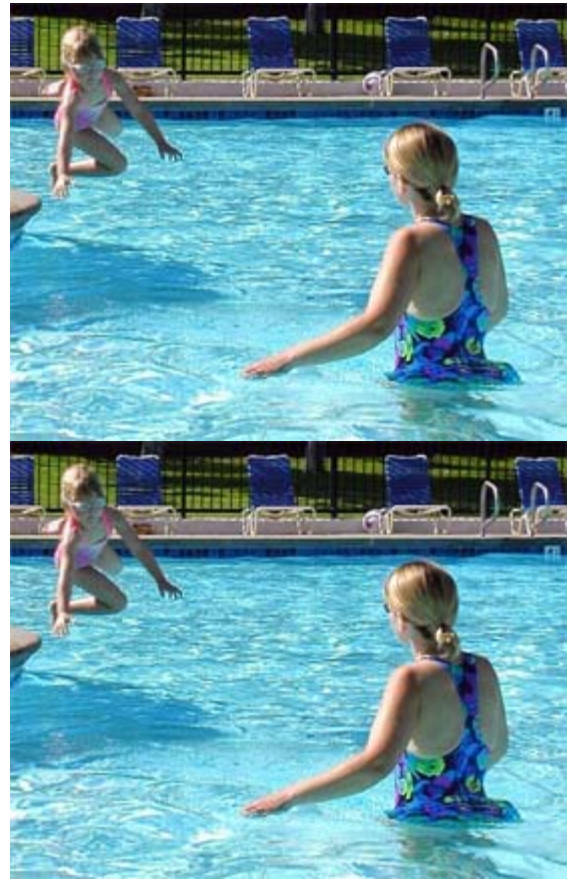
Next Page ->

Acumen Journal: PostScript Tech 18

```
<<    /ImageType 1
      ...
      /DataSource     {
           ImageData i get
           /i i 1 add store
      } bind
      ...
>> image
```

PaintProc calls the *image* operator, of course; as we said earlier, we shall use a *data acquisition procedure* for that operator's data source.

Our DAP gets the i^{th} string out of the *ImageData* array and then increments *i,* in preparation for the next time the procedure is executed.



Use the Form

```
JumpForm execform
0 219 translate
JumpForm execform
```

Finally, we execute our form twice, producing the results above.

Pretty cool, eh?

Next Page ->

So, Why is This Cool?

What's best about this way of getting an image into a form is that it will work on a very wide range of printers; in fact, this will work on *any* Level 2 or Level 3 printer that has enough VM to hold the data. The printer doesn't need a hard disk; it doesn't need to be LanguageLevel 3. There are no particular disadvantages. The other techniques do have their places, however:

- Saving the image data on a hard disk has the benefit that it uses little VM. The image data never resides entirely in memory, so image size is not limited by VM. Also, you could download the image data to the printer's hard disk ahead of time and have the data available to all future PostScript files on that printer. (We discussed something similar to this in the January 2002 *Journal.)*

  (While you're about it, you could save the entire form definition as a PostScript Resource on disk, but that's another article. Maybe. It's a long story. Take the Advanced PostScript class.)

- Using *ReusableStreamDecode* is, to my eye, a clean, satisfying, elegant solution to the problem. If I were writing PostScript for a known Level 3 printer with adequate VM, I would use *ReusableStreamDecode* every time.

Use our array-of-strings technique any time you're writing for unknown printers. It has the fewest restrictions.

We could make our technique cooler still by attaching a filter to the Data Acquisition Procedure, but, darn it, wouldn't you know, we're out of space.

Later, perhaps?

Return to Main Menu

Schedule of Classes, Feb – Apr 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	March 24 – 28	June 23–27 (London)
Advanced PostScript	March 3 – 7	
PostScript for Support Engineers	February 10 – 14	April 14–18
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?

London PS Classes Are Back

I've resumed teaching more-or-less-quarterly PostScript classes in London to accommodate students in Europe and the U.K. The first of these is a PostScript Foundations class scheduled for June 23–27. The exact location is yet to be determined, but it will be someplace a convenient Underground ride from Heathrow Airport.

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? Does it somehow make you yearn for distant islands that don't even have a word for "computer?"

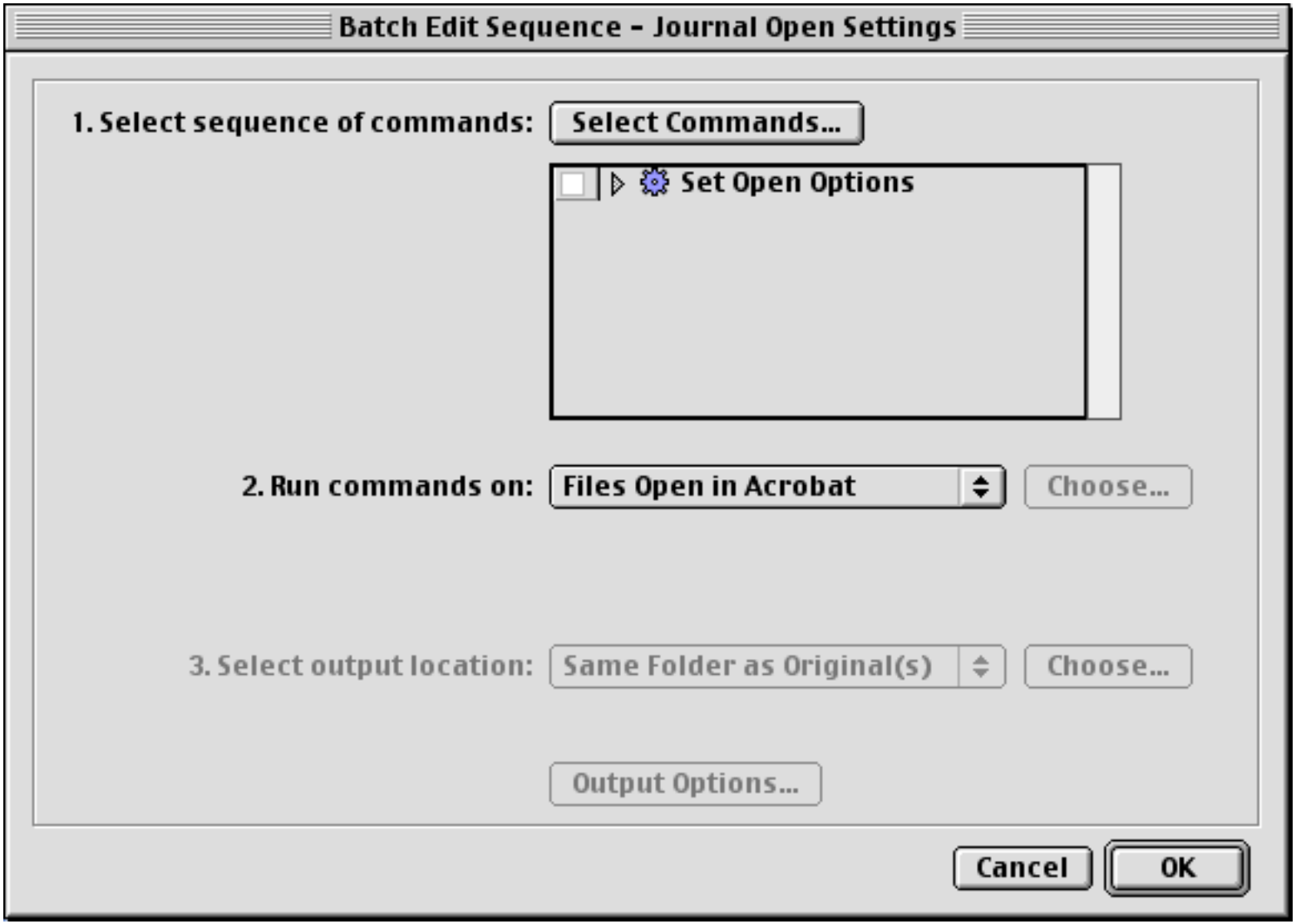
Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

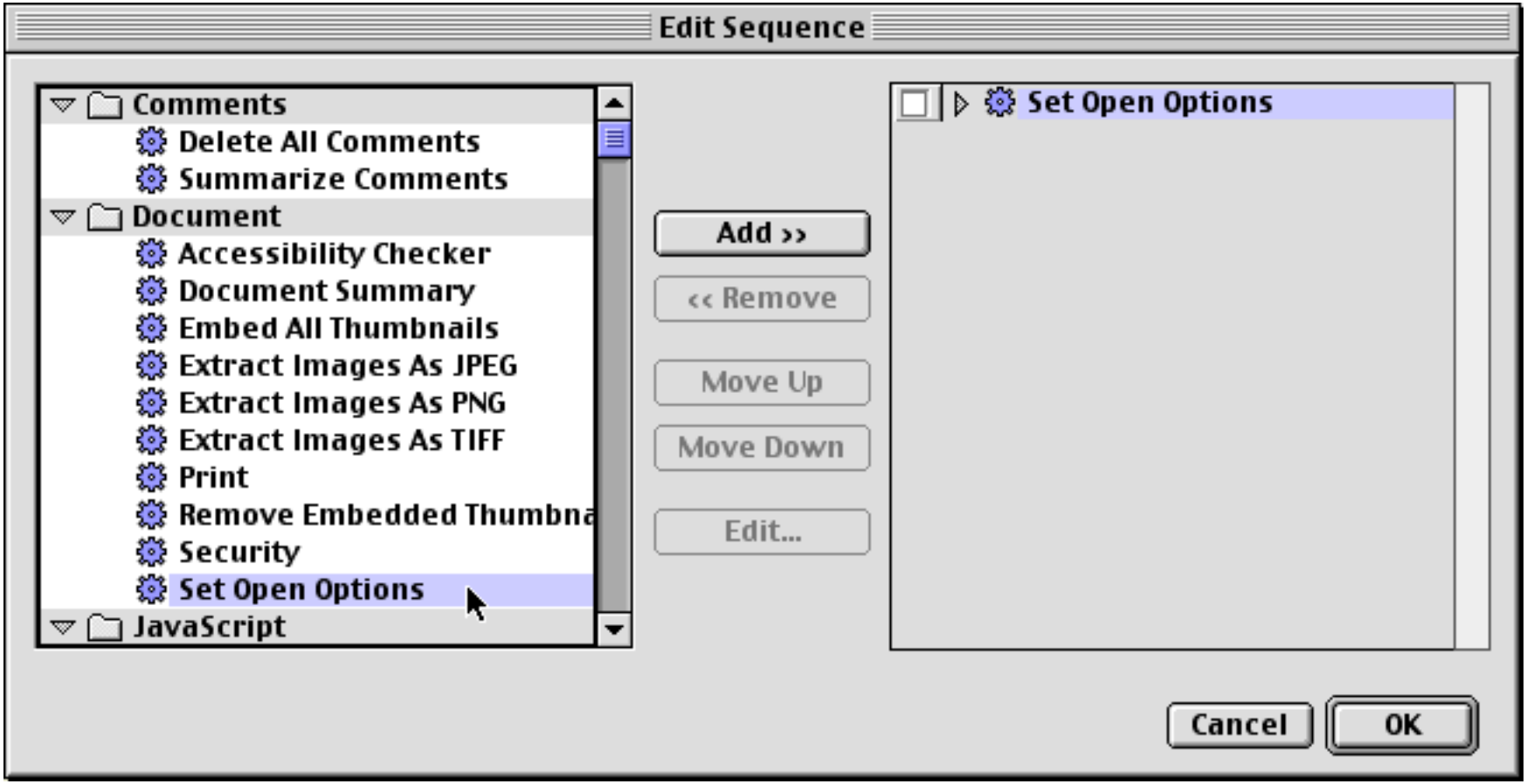
Questions and Answers. Do you have any questions about Acrobat, PDF or PostScript? Feel free to email me about. I'll answer your question if I can. (If enough people ask the same question, I can turn it into a Journal article.)

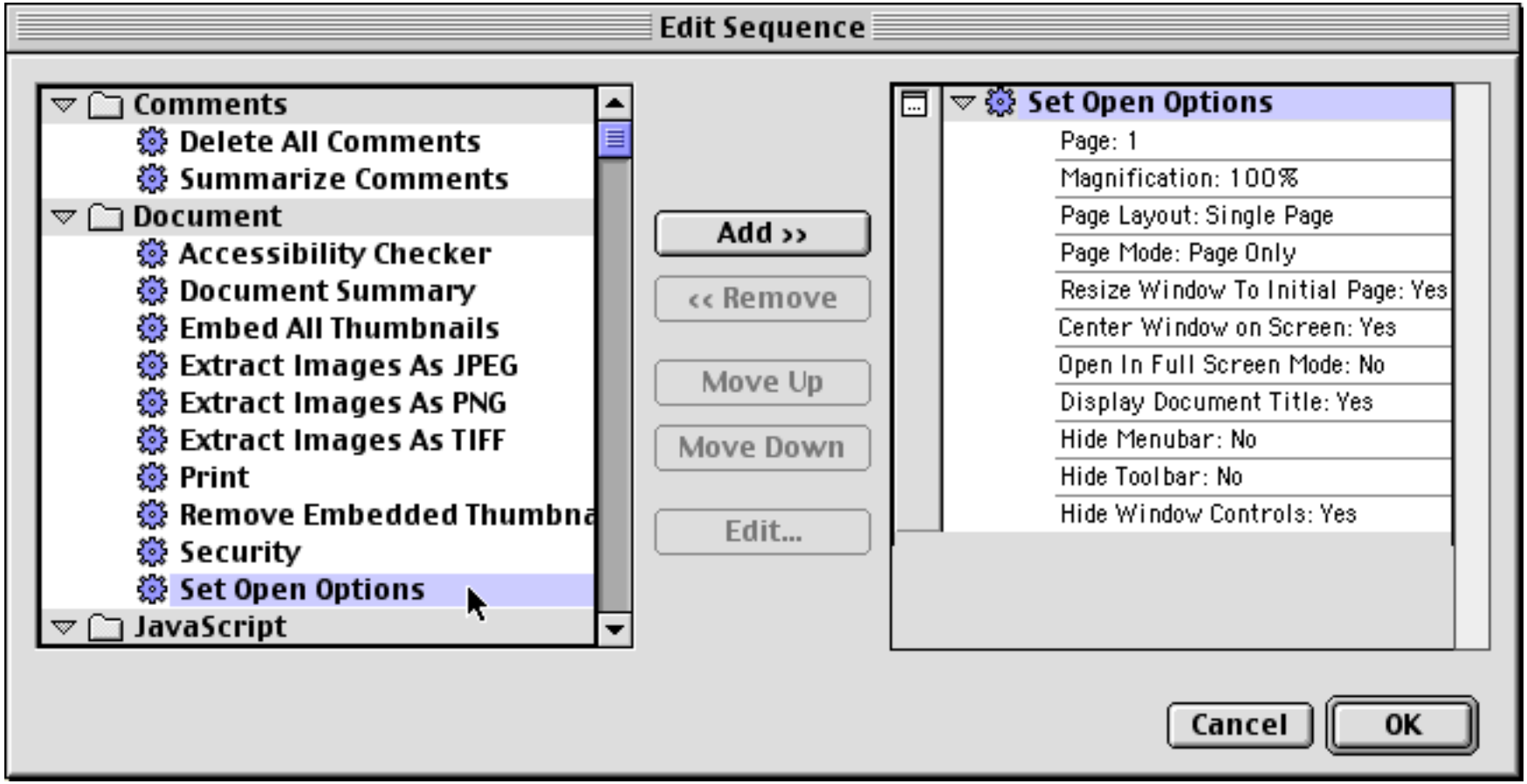
Please send any comments, questions, or problems to:

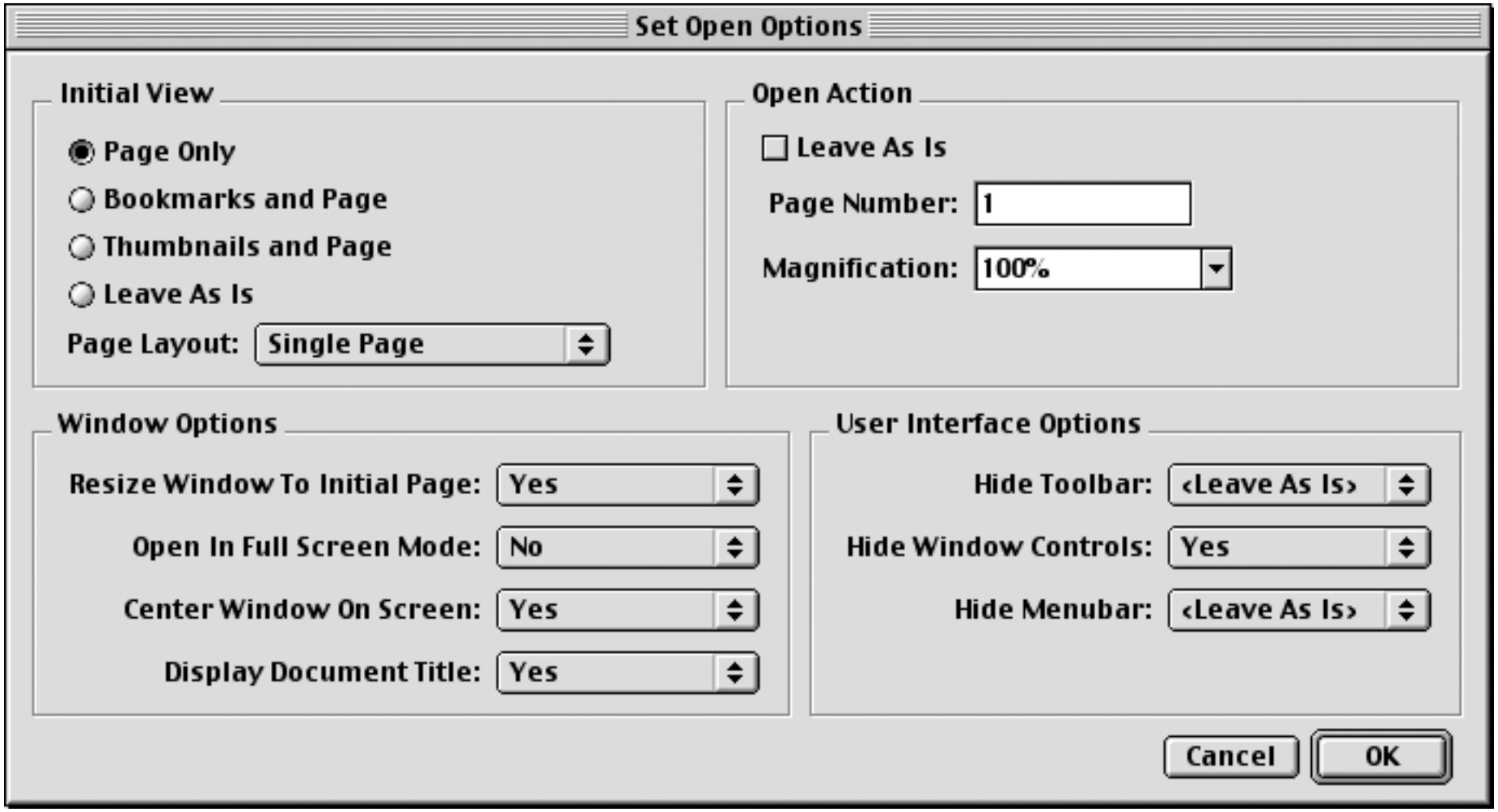
journal@acumentraining.com

Return to Menu


Batch Edit Sequence - Journal Open Settings
1. Select sequence of commands:
Select Commands...
Set Open Options
2. Run commands on:
Files Open in Acrobat
Choose...
3. Select output location:
Same Folder as Original(s)
Choose...
Output Options...
Cancel
OK


Edit Sequence
Comments
Delete All Comments
Summarize Comments
Document
Accessibility Checker
Document Summary
Embed All Thumbnails
Extract Images As JPEG
Extract Images As PNG
Extract Images As TIFF
Print
Remove Embedded Thumbna
Security
Set Open Options
JavaScript
Add ››
‹‹ Remove
Move Up
Move Down
Edit...
Set Open Options
Cancel
OK


Edit Sequence
Comments
Delete All Comments
Summarize Comments
Document
Accessibility Checker
Document Summary
Embed All Thumbnails
Extract Images As JPEG
Extract Images As PNG
Extract Images As TIFF
Print
Remove Embedded Thumbna
Security
Set Open Options
JavaScript
Add >>
<< Remove
Move Up
Move Down
Edit...
Set Open Options
Page: 1
Magnification: 100%
Page Layout: Single Page
Page Mode: Page Only
Resize Window To Initial Page: Yes
Center Window on Screen: Yes
Open In Full Screen Mode: No
Display Document Title: Yes
Hide Menubar: No
Hide Toolbar: No
Hide Window Controls: Yes
Cancel
OK


Set Open Options
Initial View
Page Only
Bookmarks and Page
Thumbnails and Page
Leave As Is
Page Layout:
Single Page
Open Action
Leave As Is
Page Number:
1
Magnification:
100%
Window Options
Resize Window To Initial Page:
Yes
Open In Full Screen Mode:
No
Center Window On Screen:
Yes
Display Document Title:
Yes
User Interface Options
Hide Toolbar:
‹Leave As Is›
Hide Window Controls:
Yes
Hide Menubar:
‹Leave As Is›
Cancel
OK