Table of Contents

Page Actions

You should have heard a "tock" sound when you came to this page. Try it again; return to the Journal's first page and come back here. You should hear it again.

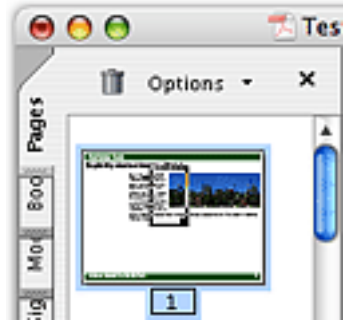
This sound is played by a *page action,* an action that is fired every time Acrobat displays this page. Acrobat has supported page actions for quite a long time; if you read either my *Acrobat Forms* or *Acrobat JavaScript* books (and why haven't you?), we discuss page actions fully and use them in several of the examples.

However, those books were written in the Acrobat 5 era. Acrobats 6 and 7 page actions work substantially the same way, but access to them has been moved to a harder-to-find location. Since one of the most common emailed questions I get regarding my books is "Where the heck did page actions go?", I thought we should review page actions in an Acrobat 6 and 7 context.

Next page ->>

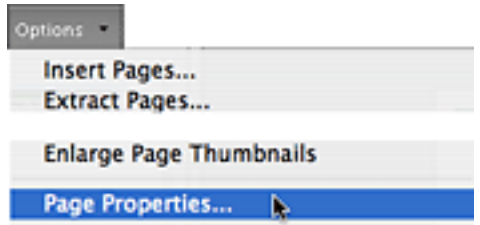
Making a Page Action

In Acrobats 6 and 7, you attach an action to a page through the *Pages* navigation pane on the left side of the document window. Do the following:



1. Select the target page in the *Pages* pane.

2. Select *Properties* from the *Options* drop-down menu at the top of the Pages pane. (Alternatively, you may right-click on the page and select *Properties* from the resulting contextual menu.)

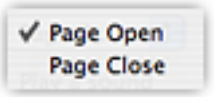
   

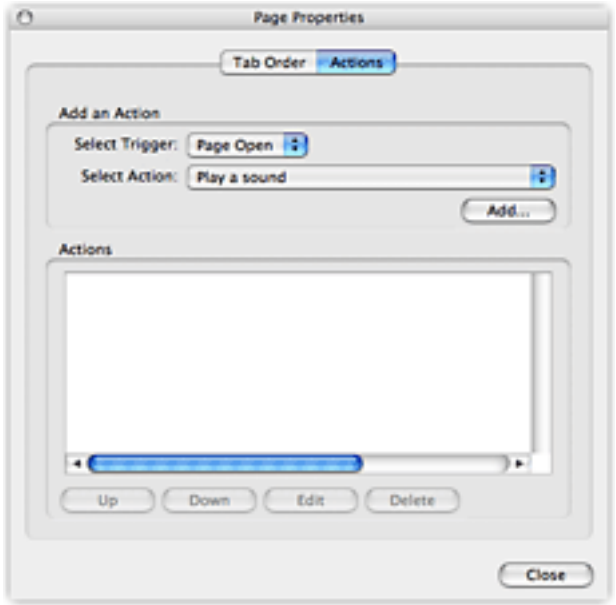
   Acrobat will present you with the *Page Properties* dialog box, below right.

3. Click on the *Actions* tab.

   You will now be looking at the a set of controls similar to those for assigning actions to a button or form field.

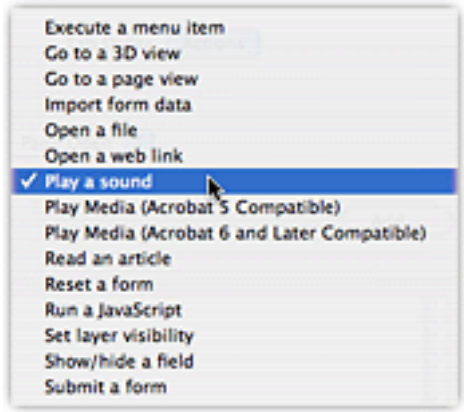
4. Use the *Select Trigger* pop-up menu to specify whether the action should trigger when the user enters the page or leaves the page.



Next page ->

5. Select an action type in the *Select action* pop-up menu.

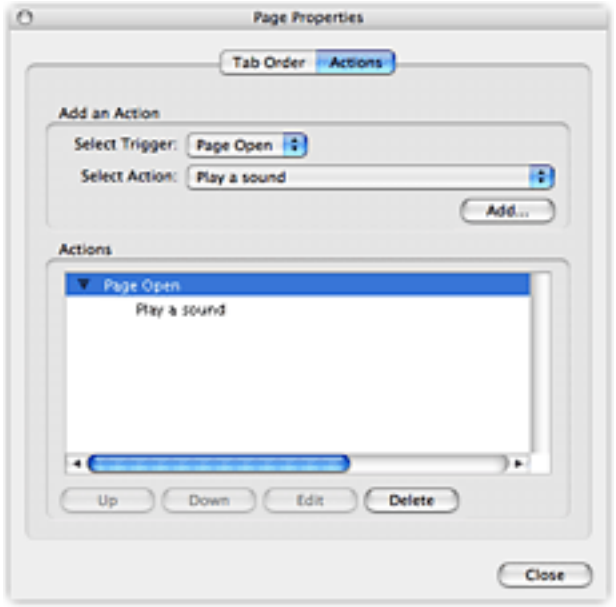


   Note that you have available to you all of the actions that Acrobat supports; any action you can attach to a button or other form field, you can also attach to a page.

   For this article's first page, I selected "Play a sound."

6. Click on the *Add…* button.

   Acrobat will present you with a dialog box appropriate to the action you selected. In the case of *Play a sound,* Acrobat has us select an appropriate sound file. (Acrobat supports any sound file format that is known to QuickTime, including wav and aif; the sound you select is converted to an internal PDF format and embedded in the PDF file.)



   When you return from specifying the parameters for your action (selecting the sound file in our case), the *Actions* panel will reflect your choice of action.

   You may add additional actions to the page by repeating steps 5 and 6 as often as you like.

Next page ->>

7. Click on the *Close* button.

   Acrobat will return you to the document page.

That's it; we have now attached an action to the page.

A View Counter

Let's look at a second example. In the left margin of this page is a box that tracks how many times this page has been viewed. Check it out: go to the next page and return here; the number will have incremented. Furthermore, if you close this document, reopen it, and return to this page, the number will increment again; the count survives across closing and reopening the file.

Page count: 4

In this case, our page action is a JavaScript that increments a global variable. The variable in question is a property of the Acrobat *global* object. To follow this discussion, you should first reread the article on the global object in the August 2004 Acumen Journal; I shall assume here that you remember how to manipulate the global object.

By the way...

Note that the counter above is actually a button, rather than a text field. This made it easier to attach a "reset counter" script to it.

I attached the page action to this page in exactly the way I described earlier. The only difference is that the page action is now "Run a JavaScript," rather than "Play sound." When you click on the *Add* button (step 6 on the previous page), Acrobat presents you with a text editor into which you can type your JavaScript.

Next page ->

The JavaScript

Our page JavaScript increments the value of a global property named *gPageCounter* and then changes the label of a button named *btnCount,* which is the number you see on the page.

Specifically, JavaScript does the following:

- Check to see if the property *global.gPageCounter* already exists.
  - If not, create it.
  - If so, increment it.
- Change the label of *btnCount* to be "Page count: " + global.gPageCounter.

Here's the page JavaScript code:

```
if (global.gPageCounter == null) {   % Is property absent?
     global.gPageCounter = 1          % Yes: create it...
     global.setPersistent("gPageCounter", true) % & make it persistent
}
else
     global.gPageCounter++  % If property exists, increment it

var f = this.getField("btnCount")    % Get a reference to the button
f.buttonSetCaption("Page count: " + global.gPageCounter) % Set caption
```

Pretty easy, actually. Let's look at it in some detail.

Next page ->

Acumen Journal: Acrobat User 6

Step by step

```
if (global.gPageCounter == null)  {
```

If *global.gPageCounter* doesn't exist, its value as seen by JavaScript will be "null." We check for this case and create the property if necessary.

```
    global.gPageCounter = 1
    global.setPersistent("gPageCounter", true)
}
```

You remember from the earlier article that you add a property to the *global* object by simply assigning the property a value, *1* in this case.

In addition, we use the *setPersistent* method to let Acrobat know that this property should survive the closing of the document. (If we didn't call *setPersistent,* the property would cease to exist when the document closed; the count would revert to *1*.)

```
else
    global.gPageCounter++
```

If *global.gPageCounter* exists (that is, if it wasn't null), we simply increment its value with the double-plus operator.

```
var f = this.getField("btnCount")
```

Now we get a reference to the button field that displays our view count.

```
f.buttonSetCaption("Page count: " + global.gPageCounter)
```

Finally, we set the caption of the button to something useful for the user.

Next page ->

You will notice, if you have read my earlier books, that page actions in Acrobat 6 and 7 are fundamentally the same as they were in Acrobat 5; the access to them has changed significantly, though not unreasonably. (Many people complain that accessing page actions through the Page navigation pane is counter-intuitive, since we tend to think of a navigation pane as being for, well, navigating the document. True, perhaps, but I must admit I can't immediately think of any better place to put page actions.)

Return to Main Menu

Explicitly Masked Images, Part 1

A couple of issues back (August 2004, to be precise), we discussed *color key image masking,* in which we print an image, specifying that certain ranges of colors be left unpainted. The two images at right, for example, are identical, except that in the lower image the blue pixels are left unpainted.

PostScript languagelevel 3 supports a second type of image masking, however: *explicit masking*. In this case, we supply, in addition to the image data, a second, 1-bit image that specifies what part of the image should be painted. This 1-bit image is referred to as a *mask* and the combined image is a *masked image*.

For example, below a masked image is created from a photograph and a 1-bit text image.

This issue and next we shall see how to do this.

Next page ->

The *image* Operator

Let's start with a very brief review of how images are printed in PostScript. We discussed this in your earlier PostScript classes, so you will find full coverage of this topic in your student notes. (Please tell me you still have your students notes.)

Images are printed with the PostScript *image* operator. In Level 2 and 3, this operator takes a single dictionary—an "image dictionary"—as its argument:

```
<<   /ImageType     1
     /Width         450
     /Height        338
     /BitsPerComponent  8
     /ImageMatrix [ 450 0 0 -338 0 338 ]
     /DataSource    currentfile /ASCIIHexDecode filter
     /Decode        [ 0 1 0 1 0 1 ]
>> image
4C65BF4...
```

Briefly, the necessary key-value pairs are these:

/ImageType The type of image; normal, scanned images are of type 1.

/Width
/Height The number of pixels in each scanline and the number of scanlines in the image.

/BitsPerComponent
The number of bits associated with each color component in the image data. Legal values are 1, 2, 4, 8, and 12.

Next page ->

A value of *8* for an RGB image indicates that each red, green, and blue will consist of an 8-bit value.

/Decode Defines the mapping of data values (varying from 0 to 255, say) to color values. The array contains a pair of color values for each color component in the image's data; an RGB image will have three pairs of numbers. Each pair of numbers indicates the color value that corresponds to the smallest and largest data values.

Thus, for an 8-bit image, a *Decode* pair *[0 1]* indicates that data values *0* to *255* should be mapped to RGB color values *0* to *1*.

/DataSource The source of the image data. It may be a file object, a string, or a data acquisition procedure. (For the last, I refer you to your PostScript notes; it's a long story.)

In our case, the DataSource is *currentfile* with the ASCIIHexDecode filter attached; the *image* operator will read Hexadecimal image data in-line with the PostScript code.

Incoming image data is interpreted in terms of the current color space. To print an RGB image, you would need to set the color space to *DeviceRGB* before calling *image*.

Next page ->

Acumen Journal: PostScript Tech 11

/ImageMatrix

A transformation matrix that specifies the size and position of the final printed image, transforming the printed image's position in User Space back to the original data in Image Space. It's a long story; look in your student notes.

As a help, the image matrix will almost always be:

```
[ width 0 0 -height 0 height ]
```

where *width* and *height* refer to the width and height of the image data in pixels, not the size of the printed image. This image matrix prints the image as a 1-unit square at the origin. We need to precede our call to *image* with a *translate* (moving the origin to our desired location for the image) and *scale* (resizing a 1-unit square to the size we want for the printed image).

Does this all sound familiar?

Next page ->

Acumen Journal: PostScript Tech 12

Masked Images

Explicitly masked images have three components:

- An *image dictionary,* a standard ImageType 1 image dictionary that draws the image.

- A *mask dictionary,* a 1-bit *ImageType 1* dictionary that supplies mask data, indicating what parts of the image should be printed.

- A *masked image dictionary,* an *ImageType 3* dictionary that ties together the image and mask dictionaries. This is the dictionary you will hand to the *image* operator.

Let's look at each of these three components in more detail.

ImageType 3 Dictionary

An image dictionary of ImageType 3 must supply three pieces of information among its key-value pairs:

- The image dictionary
- The image mask dictionary
- An indication of how the mask and image data will be integrated.

Next page ->

Acumen Journal: PostScript Tech 13

Here is a typical ImageType 3 dictionary:

```
<<   /ImageType 3
     /DataDict theImage
     /MaskDict theMask
     /InterleaveType 3
>> image
```

The key-value pairs here are as follows:

/ImageType This is a code that indicates what kind of image dictionary this is. In this case, we have a type 3 image, indicating an explicitly masked image.

/DataDict The image dictionary. This is a standard image dictionary, exactly as described at the beginning of this article.

/MaskDict The image mask dictionary. This is a nearly-standard image dictionary whose differences we shall discuss below.

/InterleaveType This is a numeric code that indicates how the mask data and image data are being supplied. You may choose between having the mask and image data interleaved in a single stream or having the mask be a completely separate image. This article will look only at this last case, whose *InterleaveType* is 3.

Next page ->

Acumen Journal: PostScript Tech 14

(If you're curious about interleaved mask data, either see the *PostScript Language Reference Manual* or take my Advanced PostScript class.)

Mask Dictionary

The *MaskDict* entry in the ImageType 3 dictionary is an *ImageType 1* dictionary that supplies the mask data. A typical mask dictionary will look something like this:

```
/theMask
<<   /ImageType 1
     /Width         50
     /Height        50
     /BitsPerComponent 1
     /Decode  [ 0 1 ]
     /ImageMatrix  [ 50  0  0  -50  0  50 ]
     /DataSource <~s8W-!s8W-!s8W-!s6'F^s8W-!s8W-!s8W
                    ...
                    s8W-!s8W,Gs8W-!s8W-!s8W-!s3^~>
>> def
```

Most of this is similar or identical to a standard ImageType 1 dictionary, but there are a few changes:

BitsPerComponent

BitsPerComponent is restricted to a value of *1* in a mask dictionary. The mask values are interpreted as either "paint" or "don't paint."

Next page ->

Acumen Journal: PostScript Tech 15

Decode In a mask dictionary, the *Decode* array contains a single pair of numbers, defining the interpretation of the 1-bit mask data. The first value in the array indicates which 1-bit value indicates "paint"; the second values indicates which value means "don't paint."

Our Decode value of *[0 1]* specifies that a value of zero will mean "paint."

DataSource DataSource has the same interpretation in a mask dictionary as in a normal image dictionary. It must be associated with a file, string, or data acquisition procedure that supplies the data. Note the way I constructed the string in the sample code:

```
/DataSource <~s8W-!s8W-!s8W-!s6'F^s8W-!s8W-!s8W
                        ...
                        s8W-!s8W,Gs8W-!s8W-!s8W-!s3^~>
```

This string is bound by <~ and ~>, delimiters that denote an *ASCII85 String*. Between the delimiters are the bytes that should be placed into the string, encoded in ASCII85; the final string will contain the *unencoded* bytes.

This way of expressing a binary string is smaller than the more-common hex string, constructed with <angle brackets>. Hex strings are double the size of the binary data; ASCII85 strings are only 125% of the binary data size.

If the masked image's InterleaveType indicates that the mask data is interleaved with the image data, then the mask dictionary will not have a *DataSource* entry; the mask data will be supplied by the image dictionary's *DataSource* entry.

Next page ->

Acumen Journal: PostScript Tech 16

Image Dictionary

I don't have anything to add regarding the image dictionary to what I have said so far. The image dictionary is an entirely standard Type 1 image dictionary, exactly as described earlier.

An Example

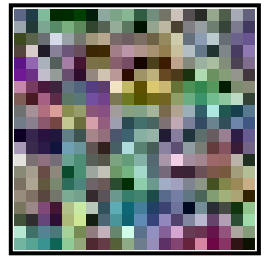
Let's look at a simple example:

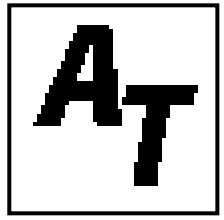
File on Server

As usual, this sample program is available on the Acumen Training Resources page. Look among the PostScript samples for *SimpleMaskedImages.ps.*

```
/theImage                % The image dictionary
<<   /ImageType     1
     /Width         20
     /Height        20
     /BitsPerComponent 8
     /Decode  [ 0 1 0 1 0 1 ]
     /ImageMatrix [ 20 0 0 -20 0 20 ]
     /DataSource <~J3Vsg2B$8a*s)Ec5QD3F*=E
                  ...
                  ^)D\bm3\<!ph%'7Z3aZ->D/.@~>
>> def

/theMask                 % The mask dictionary
<<   /ImageType 1
     /Width         50
     /Height        50
     /BitsPerComponent 1
     /Decode  [ 0 1 ]    % Zero mask values will paint
     /ImageMatrix [ 50 0 0 -50 0 50 ]
```





Next page ->

Acumen Journal: PostScript Tech 17

```
      /DataSource <~s8W-!s8W-!s8W-!s6'F^s8W-!s8
                   ...
                   s8W-!s8W,Gs8W-!s8W-!s8W-!s3^~>
>> def

100 400 translate              % Image location
-1 -1 62 62 rectstroke         % Draw a black border
60 60 scale                    % Image size

/DeviceRGB setcolorspace       % The masked image dict.
<<   /ImageType 3              % Print the masked image
     /DataDict theImage
     /MaskDict theMask
     /InterleaveType 3
>> image
```

The masked image dictionary and the mask dictionary are exactly the ones we saw earlier. The image consists of a simple 20x20 image of randomly-colored pixels.

Step-by-Step

```
/theImage
<<   /ImageType 1
     ...
>> def
```

We start by defining the image dictionary. This is in every way a standard ImageType 1 dictionary. In our case, the image is a 20x20 array of randomly-colored pixels.

Next page ->

Acumen Journal: PostScript Tech 18

```
/theMask
<<   /ImageType 1
     ...
>> def
```

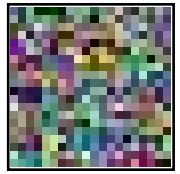
We then define the mask dictionary, which will supply the mask data. This, too, is an ImageType 1 dictionary, subject to the changes we discussed earlier.

Note that the image and mask are different dimensions (20x20 and 50x50, respectively). This is perfectly alright; the image and mask will both be mapped into the same 1-unit square at the origin, which will be converted to a final position and size by the *translate* and *scale* that precede the call to *image*.

Also note that the Decode array we used,

```
/Decode [ 0 1 ]
```

specifies that painted areas within the masked image will be denoted by *0* mask values—black pixels if we were to print the mask data as an image. Similarly, "white" areas in the mask correspond to unpainted areas in the final masked image.



Had we reversed the array,

```
/Decode [ 1 0 ]
```

we would have reversed the interpretation of "black" and "white" mask values, as at right.

Next page ->

```
100 400 translate
-1 -1 62 62 rectstroke
60 60 scale
```

The *ImageMatrix* used by our images will print the entire masked image as a unit square at the origin. Left to itself, this would be a 1/72-inch square at the lower-left corner of the paper; this is probably not what we want. So, we do the following:

- A *translate* to move the origin to the location on the page where we want the image.
- A *scale* that changes the size of a unit square from 1/72 of an inch to the size we want for the painted image.

In between, we make a call to *rectstroke* so that we get a border around our image.

```
/DeviceRGB setcolorspace
```

Because we have an RGB image, we set the current colorspace to *DeviceRGB*. This tells the *image* operator how to interpret the image data.

```
<<   /ImageType 3
     /InterleaveType 3
     /DataDict theImage
     /MaskDict theMask
>> image
```

Finally, we create an *ImageType 3* dictionary and hand it to the *image* operator.

Next page ->

Acumen Journal: PostScript Tech 20

A Problem

In the above example, the *DataSource* entries in the image and mask dictionaries were each associated with a string that contained the associated data.

```
/DataSource <~s8W-!s8W-!s8W-!s6'F^s8W-!s8...-!s3^~>
```

Unfortunately, this will not work for arbitrary images, since their data will typically not fit within a PostScript string. (These have a limit of 64k, you may recall.)

Standard images get around this by using *currentfile* for their data sources, usually with an attached filter. They then place the image data in-line with the PostScript code, as we did in our first example in this article.

```
        ...
        /DataSource currentfile /ASCIIHexDecode filter
    >> image
    4C65BF4...
```

Unfortunately, this technique will not work for masked images, since they need to simultaneously process two separate streams of data (the mask and the image).

The solution to the problem is to store both the image and mask data into VM using the *ReusableStreamDecode* filter.

However, we're out of space this month, so we'll see how to do this next time.

Return to Main Menu

Schedule of Classes, July-September 2006

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

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

Technical Classes

	PDF File Content and Structure 1	Jul 10–13		Sept 18–21
NEW!	PDF File Content and Structure 2		Aug 14–17	
	PostScript Foundations	Jul 17–21		Sept 4–8
	Variable Data PostScript			
	Advanced PostScript		Aug 7–10	
	PostScript for Support Engineers			

Course Fee The PostScript and PDF classes cost $2,000 per student.

Registration Info

Acrobat Class Schedule

Regretfully, I have suspended teaching Acrobat classes.

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: 24996 Danamaple, 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: 24996 Danamaple, Dana Point, CA 92629

Back issues

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

Return to First Page

What's New at Acumen Training?

PDF 2 Class is now complete

The tweaking of the PDF File Content and Structure 2 class is finished and, in particular, the class' topic list is now complete. The course has been taught a half-dozen times and is now running very well. The final course outline is as follows:

- Review of PDF file structure
- Determining Interiors
- Line-drawing Details
- XRef and Object Streams
- External File References
- Masked images
- Function Dictionaries
- Shading
- Rendering Parameters
- Transfer Functions
- Halftoning
- Printers' Annotations
- Marked Contents
- Form Fields
- Multibyte Fonts
- Linearized PDF
- Optional Content Groups

Most of these topics are relatively long, taking an hour or more (sometimes a *lot* more) to discuss.

For more information, see the course description 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 reflect on the brevity and futility of life?

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