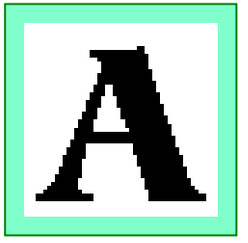
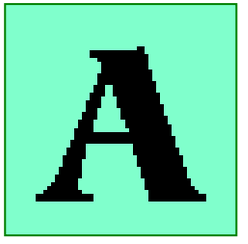
Table of Contents

Journal feedback: suggestions for articles, questions, etc.

The *imagemask* Operator



You do, of course, remember the *image* operator from your PostScript class. This is the PostScript operator that prints raster images, like the image at right. As you recall, images in PostScript are opaque, like everything else in PostScript. Thus, white areas in an image are opaque white and hide whatever they print atop; in the illustration at right, the white areas of the image obscure the green background rectangle.



PostScript has an alternative operator for printing images that I don't discuss in any of my classes: the *imagemask* operator. What is significant about this operator is that white areas are transparent; the background shows through the white spots, again as at right. This makes it perfect for such things as characters in a bitmapped font.

Compared to image

The *imagemask* operator is virtually identical to the regular *image* operator; it differs in two significant ways:

- It is limited to 1-bit data; each image sample will have a value of either *0* or *1*.
- The samples are interpreted as either painted or unpainted (rather than black or white). Painted samples will be painted with the current color; unpainted samples will allow the background to show through.

Ignoring Level 1

In this article, we are going to be discussing *imagemark* only as it exists in PostScript levels 2 and 3; we shall not discuss the Level 1 incarnation, which is left to the readers' own research.

Next Page ->>

Using *imagemask*

The *imagemask* operator, like *image,* takes a single dictionary as its argument:

```
<< imagedict >> imagemask
```

The key-value pairs that have meanings wthin this dictionary are virtually identical to those within the *image* operator's dictionary argument; briefly described, the key-value pairs are:

/ImageType *1*
This code indicates what kind of image dictionary this is. For *imagemask,* the value must always be *1.*

/Width *int*
The width of the image in samples, that is, the number of samples on a scanline.

/Height *int*
The height of the image data in samples, that is, the number of scanlines in the image.

Note that *Width* and *Height* describe the dimensions of the image *data* and are completely unrelated to the size on the page of the printed image.

/BitsPerComponent *1*
The *BitsPerComponent* entry must have a value of *1.*

/Decode *[ ]*
Decode is an array of two numbers that determines the meanings of data values *0* and *1,* that is, which value indicates "paint" and which "don't paint." There are only two possibilities:

[0 1] specifies that painted samples have a value of *0.*

[1 0] specifies that painted samples have a value of *1.*

Next Page ->>

/ImageMatrix *[w 0 0 -h 0 h]*
This is a PostScript transformation matrix that establishes the position and size of the final printed image. As you likely recall, this array of six numbers will virtually always be

```
[ w 0 0 -h 0 h ]
```

where *w* and *h* are the width and height of the image data, identical to the *Width* and *Height* dictionary entries. This array maps the image into a 1-unit square at the origin, which is not by itself terribly useful. The trick is to precede the call to *imagemask* with a *translate* and *scale* that specify the actual location and size of the printed image.

I'm certain you remember this from PostScript class.

/DataSource *fileobj | (string) | {proc}*
Finally, the *DataSource* entry supplies the image data. This is nearly always either a file object or a string, whose contents are interpreted as 1-bit image data. It is less frequently a *data acquisition procedure* that will be repeatedly called by *imagemask*. Each time it is called, the procedure must return a string containing image data.

Next Page ->>

An Example

Here's an example of the *imagemask* operator at play.

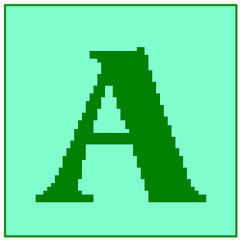
Example on Website

As usual, the sample program is available on the Acumen Training Resources page. Look for *imagemask.ps*.

```
[ 90 490 120 120 ] dup          // Draw the background rectangle...
.5 1 .8 setrgbcolor rectfill    // ... Filled with light green
0 .5 0 setrgbcolor rectstroke   // ... and stroked with dark green

0 setgray                       // Set the current color to black

100 500 translate               // The image will be printed at 100,500
100 100 scale                   // It will be printed 100 units on a side

<<  /ImageType 1
    /Width 50                   // Samples per scanline
    /Height 50                  // Scanlines in the image
    /BitsPerComponent 1         // 1-bit data (no other choice here)
    /Decode [ 0 1 ]             // 0 samples will paint; 1 will not paint
    /ImageMatrix [ 50 0 0 -50 0 50 ]    // [ w 0 0 -h 0 h ]
    /DataSource currentfile /ASCII85Decode filter    // Read data from currentfile…
                /LZWDecode filter           // ... through two filters
>> imagemask                        // Invoke imagemask, followed by the image data
J3Vsg3$]7K#&$;kY]q,L”Nu?=!&.@P!X-u’3$Q^>KPe$sNko5’p\XY>!0T5cnjja6
IfG)9fY@f@%VIG45`ib]r”1#5J’MEZq?.:a&”p^jkR’]arC`qETChaU*X;NaYU[tS
rF(3YJ%jh/Jbt&8!djtJ(‘\W9eIuLYj.H[IZK6+`[@cjpPZd*p^;&ATp’&$0_8B6W
(7AV9]/6KsT9De[:FA$(5GsLMp1AjQcVbhs2u!HV0%,8UHAeT!)$H>mi(YU*~>

showpage
```

Next Page ->

Acumen Journal: PostScript Tech 5



Step by Step This isn't particularly tricky code and I'm not going to step through it in detail. A couple parts of the program are noteworthy:

```
[ 90 490 120 120 ] dup
.5 1 .8 setrgbcolor rectfill
0 .5 0 setrgbcolor rectstroke
```

You may not have stumbled across this technique for both filling and stroking a rectangle. Remember that the *rectfill* and *rectstroke* operators both can take an array of numbers as their arguments, interpreting the array contents as *x, y, w,* and *h* for one or more rectangles.

Here we create an array with the location and dimensions of the rectangle I want to paint. I *dup* the array, then hand it successively to *rectfill* and then *rectstroke*. preceding each rectangle operator with a call to *setrgbcolor* that specifies the color of that action.

Note that the alternative, more typical, way of doing this is a bit more elaborate:

```
90 490 moveto 120 0 rlineto 0 120 rlineto -120 0 rlineto closepath
gsave .5 1 .8 setrgbcolor fill grestore
0 .5 0 setrgbcolor stroke
```

What informal testing I have done indicates that the *rectfill/rectstroke* version is consistently faster than the *gsave/grestore* method.

```
0 setgray
```

Before using *imagemask,* I set the current color to whatever I want for the image's painted pixels, black, in this case. Note that *imagemask* always paints pixels with the current color; had I not reset the color to black, the *A* would have come out in the same dark green that I used to stroke the rectangle, as at right.

Next Page ->

Acumen Journal: PostScript Tech 6

```
100 500 translate
100 100 scale
```

Remember that the location and size of the image on the page is determined by the calls to *translate* and *scale*. In this case, the lower-left corner of the image will be at *100,500* and the image will be 100 PostScript units on a side.

```
<<
     /ImageType 1
     ...
     ...
     /DataSource currentfile /ASCII85Decode filter
                  /LZWDecode filter
>> imagemask
J3Vsg3$]7K#&$;kY]q,L”Nu?=!&.@
```

Finally, we make our call to *imagemask*. The key-value pairs in the dictionary argument are as I described earlier and are relatively unremarkable. Note that the DataSource is *currentfile,* to which we have attached the *ASCII85Decode* and *LZWDecode* filters. This means that the image data, supplied in-line immediately following the invocation of *imagemask,* must have been LZW compressed and then converted to ASCII85.

```
showpage
```

Finally, we follow our image data with a call to *showpage*. We can get away with this—that is, *imagemask* doesn't interpret this PostScript line as image data—because the in-line data ends with "~>", which is the end-of-data marker for the *ASCII85* filter. The *imagemask* operator will see this two-character sequence as logical end-of-file and return control to the PostScript interpreter.

Next Page ->

Acumen Journal: PostScript Tech 7

Messing With Decode

The interpretation of the bits in the data is determined by the *Decode* array. The first entry in the array indicates the 1-bit value that should be painted; the second entry is the value that should be left unpainted. Thus, reversing the *Decode* array entries in our example (making it *[1 0])* reverses the interpretation of the bits in the image, as at right.

If you use the same value for both entries in *Decode*—that is, both *0* or both *1*—you get indeterminate behavior. Distiller treats the situation as though you have used *[0 1];* zero pixels are painted and non-zero pixels are unpainted. Ghostscript simply doesn't paint the image, which is reasonable, if a bit boring.

Return to Main Menu

Creating a "Print-Only" PDF Document

Someone (not an ex-student) once asked if I could send him a chapter from my PDF 1 student notes. This always poses a bit of a problem for me. The situation was such that the request was completely reasonable; he had asked me a question regarding something that I discuss in the PDF class and the easiest way to get the information to him would be to send him a copy of the appropriate part of my student notes.

Still, I'm never happy to send electronic copies of my notes; those are, after all, part of what I sell for a living: engineering classes and their associated student materials.

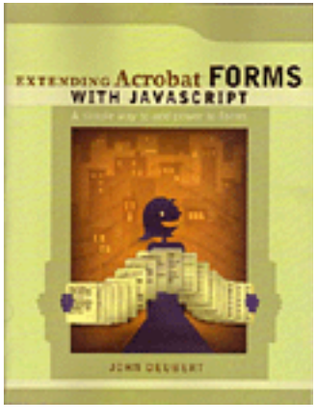
Ignoring questions of whether or not my concern is valid, the situation led me to the question: how could I send the person a PDF file that can be printed once as a personal reference, but then becomes unusable.

This month, we shall look at my solution to the problem. As with many JavaScript-based security solutions in Acrobat, it's not foolproof; its main effect is to make a crystal clear statement that you don't the file to be used once it has been printed.

Required Reading

I am going to assume that you have recently read *The Acrobat Global Object* in the August 2004 *Acumen Journal*. This describes the Acrobat JavaScript *global* object, upon which our print-only documents depend.

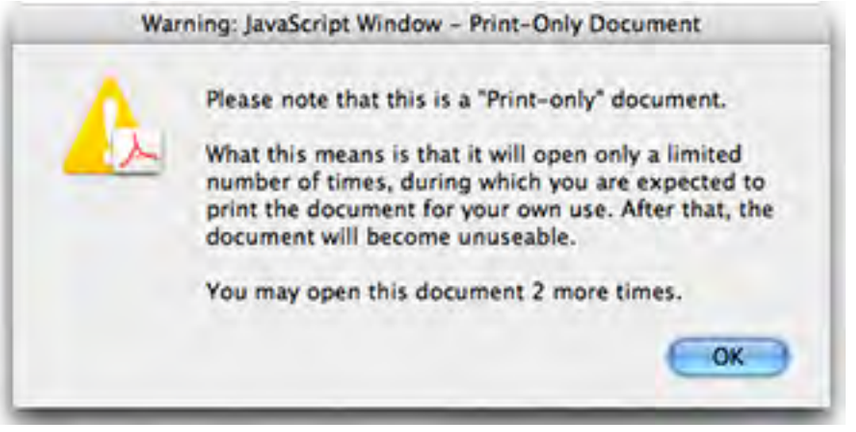
I also assume you have either read my book *Extending Acrobat Forms With JavaScript* or have equivalent experience. In particular, you will need to know how to attach a document JavaScript to a PDF file within Acrobat; happily, this is reviewed in the same August 2004 *Journal* article.



Next Page ->

Overview

A print-only PDF file allows itself to be opened only a small number of times before becoming useless. The intent is that the user will print the file during its "openable" period.



Each time it is opened, the document displays a warning, telling the user how many more times the document may be opened.

The print-only PDF file is built around a Document JavaScript, that is, a JavaScript that executes every time the file is opened. Our JavaScript does the following:

- Check for the existence of a global variable that tracks the number of times the file has been opened.
  - If the variable doesn't exist, create it and initialize it to the maximon number of times the document may be opened.
  - If it does exist, decrement it.
- Display an alert appropriate to the number of times the document may yet be opened.

  If the document has expired, say so. Otherwise, display the number of times the document is still openable, as above right.
- If the document has expired, render it unusable by doing the following:
  - Place a full-page, white, FreeText annotation on every page, hiding the pages' contents from view.
  - "Flatten" the document, converting all of the full-page annotations to regular PDF white-filled rectangles.

This isn't proof against a determined, sophisticated user, but it will slow 'em down a bit.

Next Page ->

The JavaScript

Here is the (edited) JavaScript that does the deed. This is necessarily edited for brevity; the full JavaScript is on the Acumen Training Resources page, as usual (see sidebar, left).

File on Website

This article's JavaScript is available on the Acumen Training Resources page. Look for the file *printonly.js*.

```
// First, we check our global variable, jrd_Counter001
if (global.jrd_Counter001 == null)  {    // If it doesn't exist...
     global.jrd_Counter001 = 3           // ...create it.
     global.setPersistent("jrd_Counter001", true)
}
else                                     // Otherwise...
     global.jrd_Counter001--             // ...decrement it

// localCount is a local variable whose value is the same as our global counter.
var localCount = global.jrd_Counter001  // This seemed convenient for presentation

if (localCount > 0)    {    // Haven't exceeded the max? Present an alert & continue...
     var s = "Please note that this is a \"Print-only\" document.\n\n"
     ...
     ... // Here we assemble the string s with a message appropriate to the number of
     ... // times the doc is still openable; see the full file for the details.
     ...
     // Now display the alert.
     app.alert({ cTitle: "Print-Only Document",
                 cMsg: s,
                 nIcon: 1
               })
}
```

Next Page ->

Acumen Journal: Acrobat User 11

```
else {    // If the counter has wound down to zero, kill the document.
     // First, tell the user about it.
     app.alert({   cTitle: “Document Expired”,
                   cMsg: “This print-only document has expired.”,
                   cIcon: 0
               })
     // Then, kill it.
     for (i = 0; i < this.numPages; i++){    // For each page in the document...
          this.addAnnot({                    // Add a FreeText annotation
               page: i,
               type: “FreeText”,
               rect: [ 0, 0, 612, 730 ], // Cover the entire page
               alignment: 1,
               contents: “ “,             // No message, though we could taunt them
               fillColor: color.white,
               textSize: 30,
               width: 0,
               readOnly: true
          })
     }

     this.flattenPages()              // Finally, convert the annotations to PDF artwork
}
```

Next Page ->>

Step-by-Step Let's see how this JavaScript works.

```
if (global.jrd_Counter001 == null)  {
     global.jrd_Counter001 = 3
     global.setPersistent("jrd_Counter001", true)
}
```

The first thing our script does is check to see whether there exists a global variable (you *did* read the article on the Acrobat *global* object, didn't you?) named *jrd_Counter001*. The name is arbitrary; I wanted something that was likely to be unique so I don't stomp on some existing global variable.

This variable will hold a count of the number of times the document may yet be opened. If the variable doesn't exist (that is, if it is null), then we shall create, setting its value to *3*. Of course, this can be whatever you wish for the maximum number of times the document can be opened.

We also make the variable persistent, so that it will continue to exist beyond the present Acrobat session.

```
else
     global.jrd_Counter001--
```

If *jrd_Counter001* exists, then we decrement it; when the counter reaches *0,* the document will have expired.

```
var localCount = global.jrd_Counter001
```

We create a local variable named *localCount,* which is set to the current, probably-now-decremented, value of *jrd_Counter001*. This is purely for presentation purposes in this article; the code looked cleaner with a variable name shorter than "global.jrd_Counter001."

Next Page ->

Acumen Journal: Acrobat User 13

```
if (localCount > 0)   {
     var s = "Please note that this is a \"Print-only\" document.\n\n"
     ...
     ...
```

If *localCount* is greater than zero (that is, if the document hasn't yet expired), we shall want to present some message to the user explaning that he or she has only a few more chances to print the file. We start by assembling a string to be presented to the end user. I removed most of this code from presentation here, because it isn't particularly interesting and is easy to understand by inspection in the *js* file. The result of the missing code is that the string *s* will contain the message telling the user what's going on.

```
     app.alert({ cTitle: "Print-Only Document",
                 cMsg: s,
                 nIcon: 1
               })
}
```

The *if* clause ends by presenting the user with an alert displaying the string *s*.

```
else {
     app.alert({   cTitle: "Document Expired",
                   cMsg: "This print-only document has expired.",
                   cIcon: 0
               })
```

If the document has expired, we first display an alert informing the user of that fact.

Next Page ->

Acumen Journal: Acrobat User 14

```
for (i = 0; i < this.numPages; i++){
     ...
     ...
}
```

We then execute a *for* loop that steps through each page in the document. For each page, we do the following:

```
this.addAnnot({
     page: i,
     type: "FreeText",
     rect: [ 0, 0, 612, 730 ],
     alignment: 1,
     contents: " ",
     fillColor: color.white,
     textSize: 30,
     width: 0,
     readOnly: true
})
```

This call to the Document object's *addAnnot* method places a white, full-page, FreeText annotation on the current page, effectively covering page's contents. Note that you might want to change the value of the *contents* property, placing some visible text in the annotation; something like "This print-only document has expired" would do nicely.

```
this.flattenPages()
```

Upon exiting the loop, we execute the Document object's *flattenPages* method, which converts all of the annotations within the document into PDF artwork. This is necessary, of course, because it would be otherwise very easy for the user to remove the FreeText annotations from the pages; converting them all to PDF rectangles, now just part of the pages' contents, makes them much more difficult to remove.

Next Page ->

Acumen Journal: Acrobat User 15

Limitations

There are two obvious limitations to this implementation of a print-only document:

- When the document expires, the page contents are not actually removed; they are simply covered up with a white rectangle. It is perfectly possible for a sophisticated Acrobat user to remove these rectangles. In my mind, the purpose of the JavaScript is to make it perfectly clear to the user that I don't want them to use the electronic version of the document once it has been printed. I can't prevent them from circumventing my wishes in the matter, but they will take a *very* serious hit to their karma.
- The global counter on which the JavaScript depends lives on the user's computer. If the user moves the document to another computer, he or she will be able to reopen the document another three times. Thus, the document actually can be opened three times per computer.

These limitations are not fixable using JavaScript. The secure answer to the problem would be to write an Acrobat plug-in that will actually remove the document's contents after the document has expired. I'll leave this as an Exercise for the Student.

Return to Main Menu

Schedule of Classes, September–November 2007

Following are the dates of Acumen Training's upcoming PostScript and PDF classes. Clicking on a class name will take you to the description of that class on the Acumen training website. These classes are taught in Orange County, California and on-site at corporate sites world-wide. See the Acumen Training web site for more information.

PDF Courses

PDF 1: File Content and Structure		Oct 1–4	
PDF 2: Advanced File Content		Oct 8–11	
Support Engineers' PDF			Nov 15–16

PostScript Courses

PostScript Foundations		Oct 22–26	
Advanced PostScript			
Variable Data PostScript			Nov 5–9
Troubleshooting PostScript	Sept 18–20		Nov 12–14

Course Fee Classes cost $2,000 per student, except for *Troubleshooting PostScript* and *Support Engineers' PDF,* which are $1,500 per student. There is a 10% discount for signing up three or more students. If you have four or more students that need to take a class, it will almost certainly be cheaper to arrange an on-site class.

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: 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: www.acumentraining.com/register.html

email: john@acumentraining.com

telephone: 949-248-1241

mail: 24996 Danamaple, Dana Point, CA 92629

On-Site Classes

Information regarding classes on corporate sites is available at www.acumentraining.com/Onsite.html. These courses are taught throughout the world; for additional information on classes outside the United States, go to www.acumentraining.com/OnsitesWorldWide.html.

Back issues

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

Return to Main Menu

What's New at Acumen Training?

Support Engineers' PDF is Available

The first *Support Engineers' PDF* class is scheduled for November 2007; on-site classes are available immediately.

This two-day, hands-on, technical introduction to the PDF file format discusses the basics of the structure and contents of a PDF file, emphasizing those parts of the PDF specification most important to printed documents. The course is a good, quick introduction to PDF structure for people who need to examine and diagnose troublesome PDF files.

Note that this is a class in the PDF file structure, not the use of Adobe Acrobat. The course does examine some commercial tools that are useful in the diagnosis of PDF problems.

See the Acumen Training website for a detailed course outline.

XPS File Content and Structure

I am currently working on a new course, *XPS File Content and Structure,* a four-day, hands-on course on Microsoft's *XML Paper Specification*. The course will be modelled on the PostScript and PDF classes, emphasizing those parts of the XPS that describe marks on a page. The target audience will be printer engineers working on XML-based printers, support personnel who work with those printers, and software writers who create XML documents. The course will be available first quarter 2008. Watch the Acumen Training website for pricing and schedule of classes.

Return to Main Menu

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 seem to have been inexpertly translated from Japanese?

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:

john@acumentraining.com

Return to Main Menu