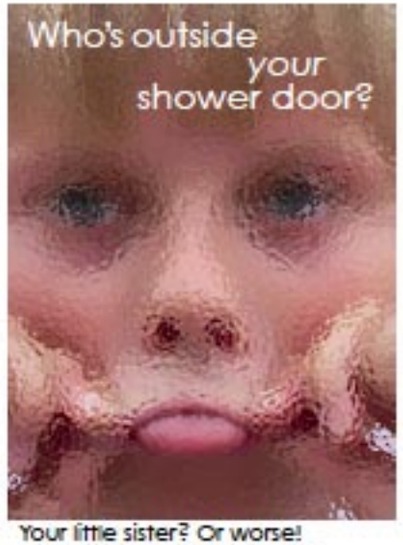
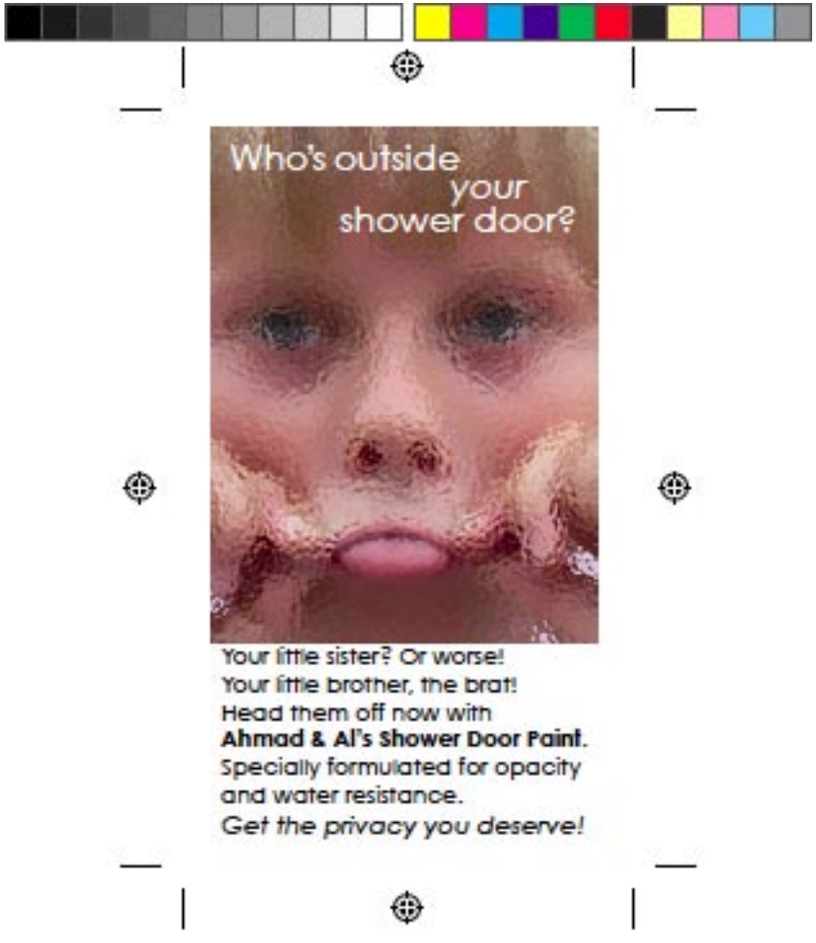
Table of Contents



Journal feedback: suggestions for articles, questions, etc.

Adding Printer Marks With Acrobat 8



Acrobat 6 and 7

This article describes how to add printer marks using Acrobat 8. However, Acrobat 7 and 6 could also add printer marks to a document.

This article's instructions to apply to Acrobat 7 more-or-less unchanged. You will find differences in the layouts of the dialog boxes, but the functionality is pretty much the same.

Acrobat 6 can add printer marks to the PDF file only at print time. The *Print* dialog box has an Advanced Options button that takes you to a dialog box that, among other things, allows you to add printer marks to the printed document.

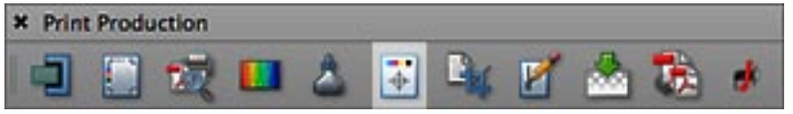
"Printer marks" is a generic term for crop marks, registration marks, color calibration bars, and other graphic features necessary to the printing of professional documents. These marks are used in the common case that a document is printed on paper (or other media) that is larger than the document's final page size. The marks reside on the paper outside the boundary of the document page.

These marks are often placed on the page by the application that is used to create the PDF document; however Acrobat can also add these marks to the pages of an existing PDF file.

This is the topic of this short Guide: how to place printer marks on a PDF page using Acrobat 8.

Next Page ->>

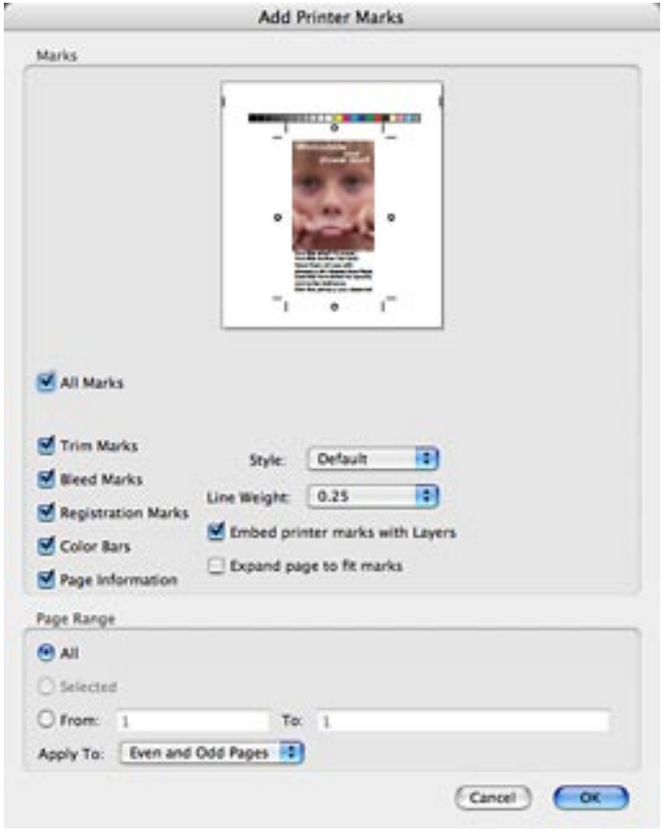
The Print Production Toolbar



You add printer marks with the *Printer Marks* tool in the *Print Production* toolbar, pictured at right. This toolbar hosts a variety of tools intended for the professional printing of Acrobat files; the tools here allow you to, among other things, soft proof the document's colors, convert colors from one color space to another (RGB to CMYK, for instance), and specify trapping parameters.

Of interest to us in this article is the *Printer Marks* tool. When you click on this tool, Acrobat presents you with the *Add printer Marks* dialog box, below right.

This dialog box is pretty straightforward; simply select the specific marks you want added to the document and the pages to which you want them added. When you click the OK button, Acrobat will add the marks to the document and return you to the document window.



Types of Printer Marks

Most of the checkboxes in this dialog box are reasonably self-explanatory; trim marks, bleed marks, registration marks, and color bars are pretty standard in the industry. (You can see these in place on the previous page's illustration.) A couple of the available selections are perhaps less immediately obvious.

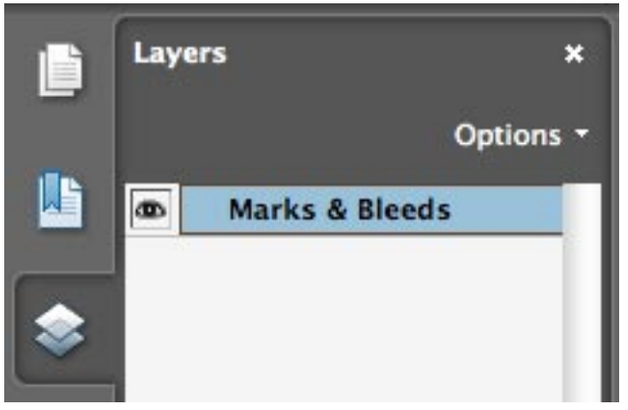
Page Information

This will place on the page the name of the PDF file, the current date and time, the page number, and the color plate.

Next Page ->

Embed printer marks with Layers

The printer marks will be placed in their own layer within the PDF file so that they can be hidden or made visible all at once using the Layers navigation pane (shown at right).



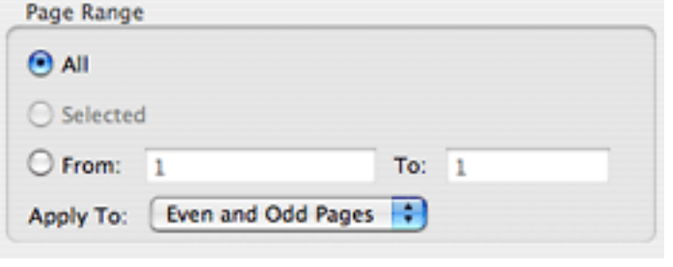
Clicking in the "eye" control next to the "Marks & Bleeds" name will toggle the visibility of the printer marks in the document.

Expand page to fit marks

Alright, so this one is pretty clear. If you select this checkbox, Acrobat will expand the page to include the printer marks. If you *don't* select this control and if your document doesn't have a trim box (more on that in a moment), then the printer marks will be placed on top of the page content, which is not usually what you want.

Selecting a Page Range

The *Add Printer Marks* dialog box provides a set of self-explanatory controls that let you specify the set of pages to which the printer marks should be added. You can select a page range and, within that range, specify whether the marks should be applied to even pages, odd pages, or both.



The Trim Box

This has all been pretty easy. We've been ignoring one point, however. Printers marks should be placed on the paper, but outside of the boundary of the document page. How does Acrobat know where the border of the document page lies on the paper?

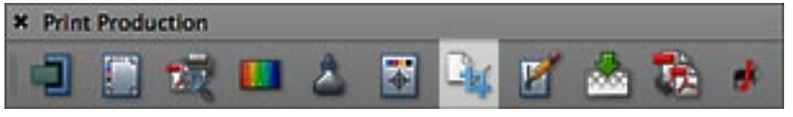
The answer is: it looks for the presence of a *Trim Box* in the PDF document.

Next Page ->

The Trim Box defines the boundaries of the document page within a PDF file. The PDF file specification defines a total of 5 rectangular regions that may be defined in a PDF file, including the *Media Box,* which defines the size of the paper on which the document is laid out, and the *Bleed Box,* which defines the area occupied by the marks on the paper; all of these are optional except the Media Box.

In particular, the Trim Box may be missing from a document, in which case Acrobat will not know where on the paper the document page lies; it will treat the entire paper as the document page, which forces Acrobat to put the printer marks on top of the page contents.

If the document page is smaller than the PDF file's paper size, Acrobat needs a Trim Box; if that box is missing, we must add it. Happily, this is pretty simple; we can add a Trim Box using the *Boxes* tool in the Print Production toolbar. Note that you must do this before you place printer marks on the page.



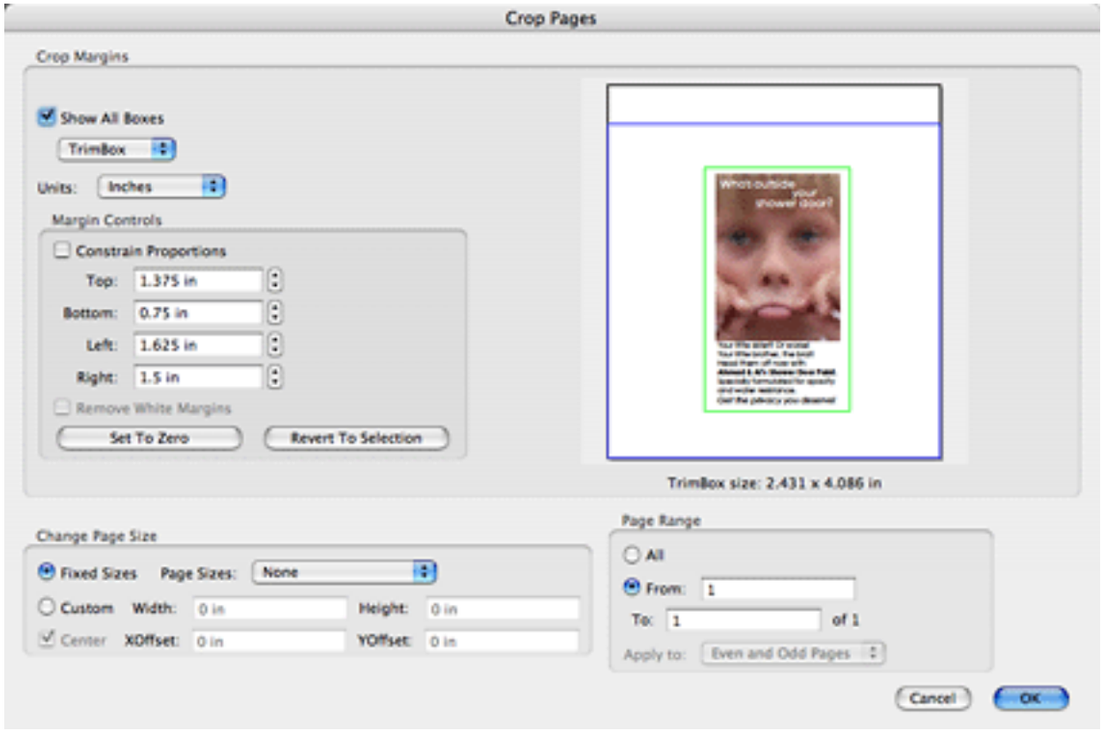
Adding a Trim Box

When you click on the Boxes tool, Acrobat presents you with the somewhat misnamed *Crop Pages* dialog box.

This dialog box allows you to specify the locations and dimensions of several of the rectangles PDF understands, including the Trim Box.

To set the Trim Box in your PDF document, do the following:

1. Select *TrimBox* in the uppermost pop-up menu.



Next Page ->

2. In the *Top*, *Bottom*, *Left*, and *Right* text boxes, enter the values you want for the "margins" that define your Trim Box. (That is, the distance from that side of the Trim Box to the edges of the paper.)

   Alternatively, you can click on the tiny up and down arrows to the right of each text box to change the value of that margin.

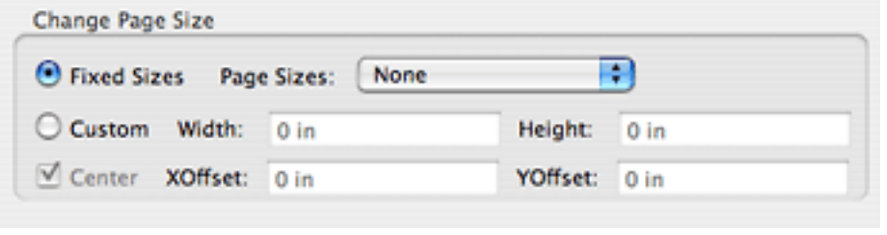
   As you change the values of the margins, Acrobat displays a green rectangle in the dialog box's preview pane, showing you where the trim box lies. At right, I've set the Trim Box to enclose the artwork in the document; note the green box surrounding the page's graphics.

   Specify the units of measure you want to use (inches, millimeters, etc.) using the *Units* pop-up menu.

Some Shortcuts

As a convenience, the *Crop Pages* dialog box provides a couple of sets of controls to select common actions related to the Trim Box:

- The *Remove White Margins* checkbox (grayed out in the illustration above), sets the Trim Box to exactly enclose all of the marks on the PDF page.
- *Change Page Size* changes the size of the paper on which the PDF document is placed. This doesn't change the Trim Box; rather it changes the paper on which the document page is placed; this allows you to replace the existing 8½" x 11" paper size with A4, for example.



Next Page ->

Crop marks are among those items it is very easy to forget when creating a document. They aren't hard to create in most design applications, but it is a nuisance to have to go back to the original InDesign file (or whatever) if you notice they are missing.

Happily, Acrobat 8 has made it very easy to add these marks.

Return to Main Menu

PostScript Resources, Part 2

Last month, we discussed the nature of PostScript resources and how to create and retrieve fonts, forms, and other data that have been stored as resources.

What we did not convincingly demonstrate is why you would want to store fonts or forms as a resource at all, rather than just as a variable. This is the question we shall answer today. The secret lies in having a RIP with a hard disk available to it.

You should read the first part of this article in the January 2007 *Journal* before proceeding with this part.

Last Month

We finished up Part 1 by creating a Form resource. This entailed creating a form dictionary and turning it into a resource of type Form with a call to *defineresource:*

```
/SquareForm   <<
     /FormType 1
     /BBox [ 0 0 100 100 ]
     /Matrix [ 1 0 0 1 0 0 ]
     /PaintProc { 0 0 100 100 rectfill } bind
>> /Form defineresource
```

We could then retrieve the form dictionary for use with *execform* by calling *findresource:*

```
/SquareForm   /Form   findresource   execform
```

To make this Form resource (or any resource) truly useful, we should store the resource definition on the RIP's hard disk.

Next Page ->

Acumen Journal: PostScript Tech 8

Disk-Based Resources

The PostScript *findresource* operator (and its older, more-specific cousin, *findfont)* looks for a requested resource in all the places that resources may be stored on the specific PostScript device. In particular, if the resource is not found in VM, then *findresource* will do the following, assuming that the RIP has a hard disk available to it:

- Algorithmically derive a file name from the names of the requested resource and the name of the resource category ("Optima" and "Font," for example).
- Look for a file with that name.
- If the file exists, execute the contents of the file.

The file is presumed to contain the PostScript code that creates the resource.

Thus, if you ask for a Form named SquareForm with a call to *findresource:*

```
/SquareForm /Form findresource
```

The RIP will examine VM to see if the form is available there; if not, the RIP will derive a file name from the combination of the category and resource names (probably something like "/resources/Form/SquareForm") and execute the file, if it exists. This file must contain the PostScript code that creates the SquareForm resource:

```
/SquareForm  <<
     /FormType 1
     /BBox [ 0 0 100 100 ]
     /Matrix [ 1 0 0 1 0 0 ]
     /PaintProc { 0 0 100 100 rectfill } bind
>> /Form defineresource
```

Once the PostScript interpreter executes this code, it will have access to the form dictionary.

Next Page ->

Acumen Journal: PostScript Tech 9

Note that the code in the file must end with a call to *defineresource,* which leaves a copy of the newly-created resource data on the stack.

To store a resource on a RIP's disk, we must somehow determine the name of the file the RIP will look for when it needs to load that resource. This will differ from one RIP to another and will not necessarily be something we can guess. (Some RIPs still use eight-dot-three file names, for example.)

Resource Filenames

Happily, the function that a RIP calls to derive a resource's filename is accessible to us; this function—a PostScript procedure—is stored in the resource Category dictionary.

The *Category* Category

At right is a table of the Regular resource categories defined by PostScript. Notice that among these is a category named *Category*. This resource type is what defines the resource categories recognized by PostScript; that is, resource categories are themselves managed as resources. Resources within the Catagory category have names like "Font," "Form," "Pattern," etc.

Regular Resource Categories

Font	Encoding	Form
Pattern	ProcSet	ColorSpace
Halftone	Category	Generic
ColorRendering	FontSet	InkParams
TrapParams	IdiomSet	

The data associated with each Category resource is a dictionary that contains procedures that dictate how the PostScript resource operators should behave when applied to a resource of that category. Particular to our purpose is a procedure named *ResourceFileName*.

```
/ResourceName (scratch) ResourceFileName  =>  (pathname)
```

This procedure takes as its arguments the name of a resource and a scratch string; it returns the pathname to the file that should contain the definition of that resource. (The resource category is implicit, since the procedure is fetched from that category's Category dictionary.)

This is the procedure that *findfont*, *selectfont,* and *findresource* use to search for a resource definition. We can use this procedure ourselves to store a resource definition where those operators will find it.

Next Page ->

Acumen Journal: PostScript Tech 10

All we need to do is:

1. Get *ResourceFileName* out of the appropriate Category resource dictionary.
2. Execute the procedure, which gives us the pathname to the target file.
3. Open a file with that pathname.
4. Write the resource definition (ending in a call to *defineresource)* into the file.
5. Close the file.

That's it; once we have done this, the resource will be available to *findfont* and *findresource* on that RIP.

Let's see how to do it.

The Code

The following code defines a procedure named *WriteResource,* which takes the names of a resource name and category as its arguments and writes the resource definition to the RIP's hard disk. The resource definition is read from *currentfile,* and so must follow the invocation of *WriteResource* in the PostScript code.

The example uses *WriteResource* to write our *SquareForm* Form resource to disk.

Next Page ->>

```
/inbuf 4096 string def
/scratch 100 string def

/WriteResource                          % Stack: /resname /restype =>
{    /Category findresource begin          % /resname
        scratch ResourceFileName           % (pathname)
        end                                % (pathname)
        (w) file                           % fileobj
        /dest exch def                     % ---
        {    currentfile inbuf readstring % (data) bool
             dest 3 -1 roll writestring    % bool
             not {exit} if                 % ---
        } loop
        dest closefile
} bind def

% The PostScript code following "WriteResource" will be written to disk.
/SquareForm /Form WriteResource
/SquareForm   <<
     /FormType 1
     /BBox [ 0 0 100 100 ]
     /Matrix [ 1 0 0 1 0 0 ]
     /PaintProc { 0 0 100 100 rectfill } bind
>> /Form defineresource
```

Having executed the above PostScript code, the RIP will forever after be able to use the *SquareForm* resource with a simple call to *findresource*. A future PostScript file could use the form thusly:

```
/SquareForm /Form findresource execform
```

Let's see how this file works in detail.

Next Page ->

Acumen Journal: PostScript Tech 12

Step by step

```
/inbuf 4096 string def
/scratch 100 string def
```

We start by defining pair of strings: *inbuf* will serve as the input buffer for reading PostScript code from *currentfile; scratch* will be the string required by the *ResourceFileName* procedure. (We could have used the same string for both purposes, thereby saving a bit of VM, but the code is clearer this way.)

```
/WriteResource          % /resname /restype =>
{    /Category findresource begin          % Stack: /resname
```

The *WriteResource* procedure starts by pushing the name *Category* onto the stack and then fetching the Category resource dictionary with a call to *findresource*. Note that *WriteResource* receives the name of the resource category ("/Form") as an argument.

The *findresource* operator returns a category dictionary on the Operand stack, which we move to the Dictionary stack with a *begin*. This allows us to access the contents of the category dictionary simply by referring to the name.

```
scratch ResourceFileName                % => (pathname)
```

At this point, the name of the resource ("/SquareForm") is still on top of the Operand stack; we push our scratch string on top of the stack and call *ResourceFileName*. This procedure, resident in the category dictionary, returns the scratch string, now holding the pathname of the file into which we should store the resource definition.

```
end
```

We are finished with the category dictionary, so we can remove it from the Dictionary stack. This line has no effect on the Operand stack, which still holds the pathname.

```
(w) file                                % => fileobj
```

Next Page ->

Acumen Journal: PostScript Tech 13

Our next step is to open the appropriate file on the RIP's hard disk. We shall do this with a call to the *file* operator:

```
(pathname) (access) file   =>   fileobj
```

The pathname is already on the stack, so we push the string *(w),* which specifies we want to open the file with write permission. The *file* operator then opens the file, returning the PostScript fileobject that represents the open file.

```
/dest exch def                          % => ---
```

We save the fileobject with the name *dest*.

```
{ ... } loop
```

Now we start our indefinite loop. Each time through this loop, we want to read a bufferful of PostScript code, write that code to *dest,* and then check to see if we are at end-of-file, exiting the loop if so.

```
currentfile inbuf readstring            % => (PS Code) bool
```

The loop starts by reading data from *currentfile* using *readstring*. The operator leaves on the stack *inbuf,* now full of PostScript code, and a boolean that will be *false* if we are at the end of the input stream.

```
dest 3 -1 roll writestring              % => bool
```

We now write the string of PostScript code to our target file with *writestring*.

```
fileobj (data) writestring   =>   ---
```

The data string is already on the stack; we push our fileobject, *dest,* on the stack and then put them in the correct order with *roll*. Our call to *writestring* then writes the PostScript code to disk.

This line leaves on the stack the boolean returned by *readstring;* remember that a false value indicates we are at the end of the input stream.

Next Page ->

Acumen Journal: PostScript Tech 14

```
    not {exit} if
} loop
```

Our loop ends by reversing the *readstring* boolean with a *not* and exiting the loop if the reversed boolean is true.

```
    dest closefile
} bind def
```

Finally, our *WriteResource* procedure ends by closing our destination file.

Now, we can use our new procedure to write resources to the RIP's hard disk.

```
/SquareForm /Form WriteResource
```

We push the names */SquareForm* (the resource name) and */Form* (the category name) onto the operand stack and execute *WriteResource*. The Procedure executes its loop, repeatedly reading the input stream and writing the data to the destination file.

The PostScript code following the invocation is the definition of the SquareForm Form resource:

```
/SquareForm  <<
    /FormType 1
    /BBox [ 0 0 100 100 ]
    /Matrix [ 1 0 0 1 0 0 ]
    /PaintProc { 0 0 100 100 rectfill } bind
>> /Form defineresource
```

The code you write to the file can do pretty much anything you wish, as long as it ends in a call to *defineresource*.

Next Page ->>

Font Downloaders

Note that our *WriteResource* procedure is the PostScript basis for a typical PostScript font downloader. You can use this procedure to write font definitions, taken from a *pfb* file, to disk and that file will be available to all future PostScript programs. You will need to extract the PostScript code from the PDF file and then save that PostScript as a Font resources. (The article "Converting PDF Files to PostScript" in the September 2002 *Acumen Journal* describes how to extract the PostScript from a PDF file.)

Return to Main Menu

Schedule of Classes, March-May 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 corporate sites world-wide. See the Acumen Training web site for more information.

PDF 1: File Content and Structure	Apr 30–May 3		Jun 18–21
PDF 2: Advanced File Content			Jun 4–7
PostScript Foundations		May 7–11	
Variable Data PostScript			
Advanced PostScript	Apr 9–12		
Troubleshooting PostScript		May 30-Jun 1	

Course Fee Classes cost $2,000 per student, except for *Troubleshooting PostScript,* which is $1,500 per student. There is a 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.

Registration Info

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: john@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?

Support Engineers' PDF

Support Engineers' PDF is a two-day, hands-on, technical introduction to the PDF file format. It 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.

Course Outline

Day 1

- PDF Data Types
- PDF Objects
- PDF File format
- The Page Tree
- Content Streams
- Simple Drawing
- Introduction to Color
- Drawing Text
- Coordinate Transforms
- Compression & Transmission Filters

Day 2

- Color and Color Spaces
- Image XObjects
- Form XObjects
- Transparency
- PDF Font Structure
- Examination of Common PDF Files
- PDF/X & PDF/A
- PDF Troubleshooting Tools

Availability

Support Engineers' PostScript will be available June 2007. Watch the Acumen Training website for pricing and schedule of classes.

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 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?

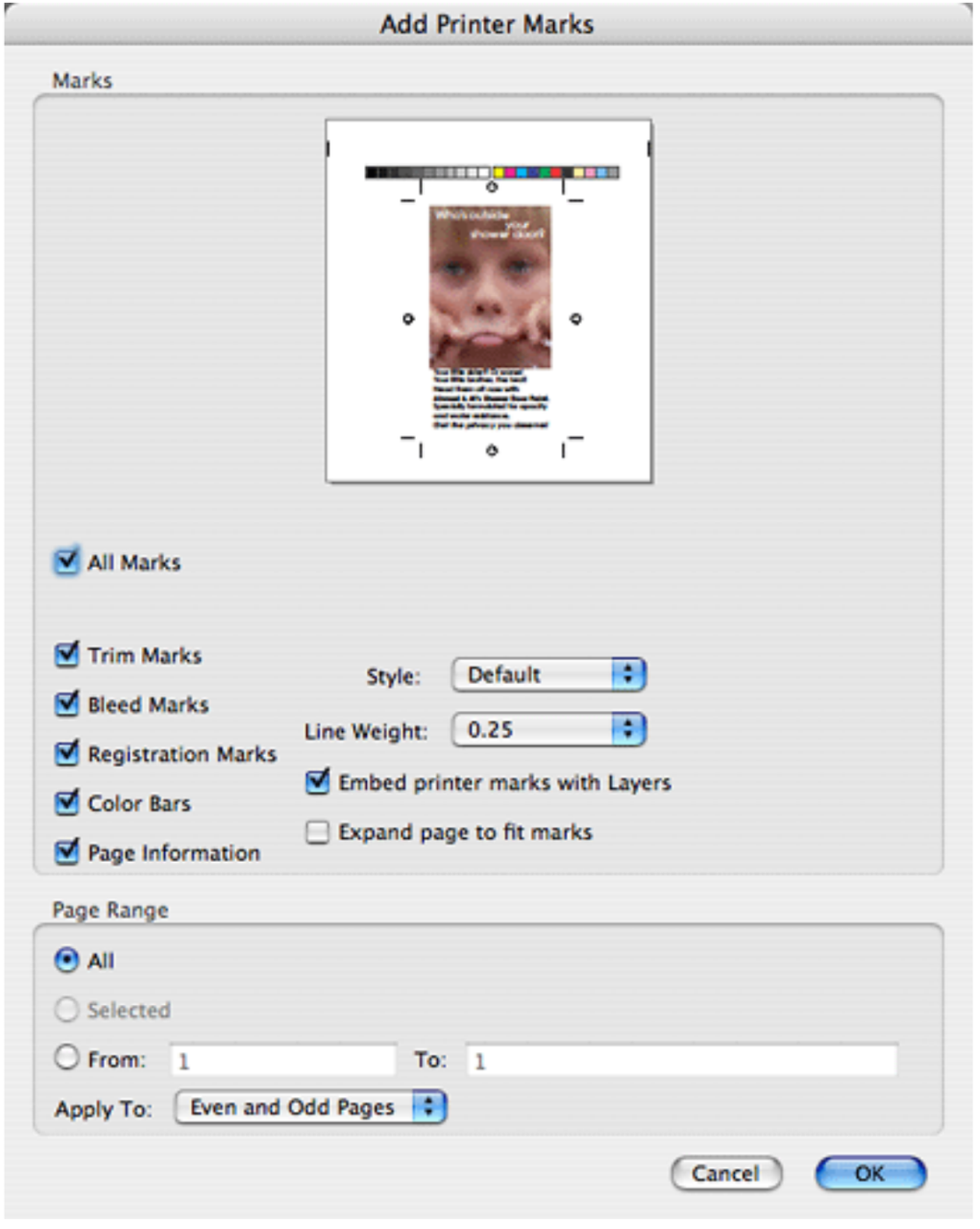
Questions and Answers. Do you have any questions about Acrobat, PDF, or PostScript? Feel free to email me about. I'll answer your question if I can. (If enough people ask the same question, I can turn it into a Journal article.)

Please send any comments, questions, or problems to:

john@acumentraining.com

Return to Main Menu

Add Printer Marks Diaog Box



Crop Pages Dialog Box