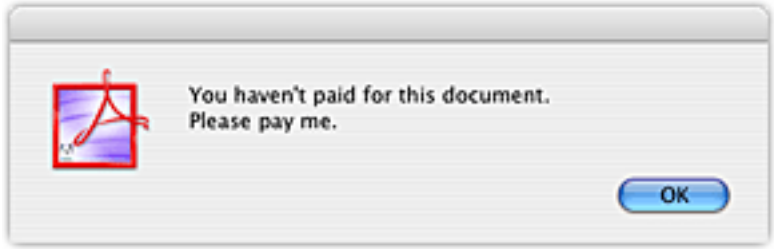
Table of Contents

JavaScript: Creating a Nagware Document, Part 2

Files on Website

As usual, the files associated with this article are available on the Acumen Training Resources page. Look for the file *Nag2.zip*.



Last issue, we started to create an Acrobat document that periodically checks to see if the user has registered and obtained a serial number and, if not, repeatedly nags the reader to do so. At the end of the previous article, we were well on our way; we had an Acrobat file that would place increasingly strident "Pay Me" messages on the page.

This month, we shall finish the job. We'll put pitiful pictures of our starving family on the page and ultimately, if even this plea is ignored, render the document unreadable.

Doing this will give us the chance to discuss JavaScript loops, something I purposely ignored in my JavaScript book.

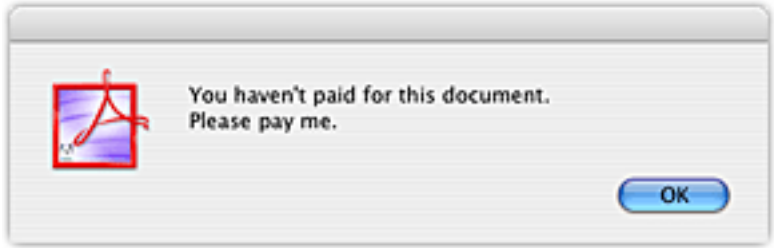
Before going any further, you should re-read the September 2004 *Journal,* to see where our nagware document stands.

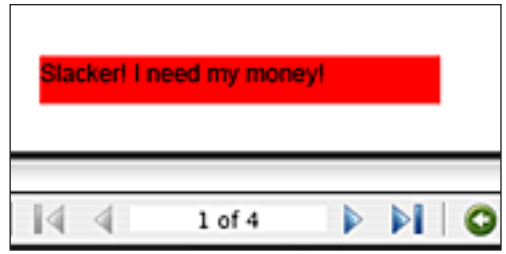
Next Page ->

Where We Are; Where We're Going

When we left off in the last issue, our nagware document did the following:

- The first two times the document is opened, it does nothing; the user is allowed to read the document unmolested.
- On the third opening, the document displays a dialog box asking for payment.
- On the fourth and fifth openings, the document adds annotations to the document's first page, again asking for payment.





This Month's Additions

This month, we are going to add two more levels of nag:

- On the sixth opening, we'll work on the reader's sympathy by presenting a photo of our poor child scavenging for food.
- On the seventh opening, we'll white out all of the pages, making the document unreadable.

We'll finish by adding a registration mechanism, removing the nags when the user supplies a serial number.

Next Page ->

The Code So Far

We implement our nags as a document JavaScript that, at this point, looks like this (this is heavily abbreviated, of course):

```
// If global.nagCount doesn't exist, then create it.
if (global.nagCount == null)    {
        global.nagCount = 1
        global.setPersistent("nagCount", true)
}
else        // Otherwise: increment it
    global.nagCount++


if (global.nagCount > 2)        // Is nagCount greater than 2?
  app.alert("You haven't paid for this document.\n Pay me.")

switch (global.nagCount)    {   // Examine global.nagCount
    case 4:                     // If it's 4, do the following:
        ...             // Add annotation at bottom of page
        break

    case 5:                     // If it's 5, do the following:
        ...             // Add annotation in middle of page
        break
}
```

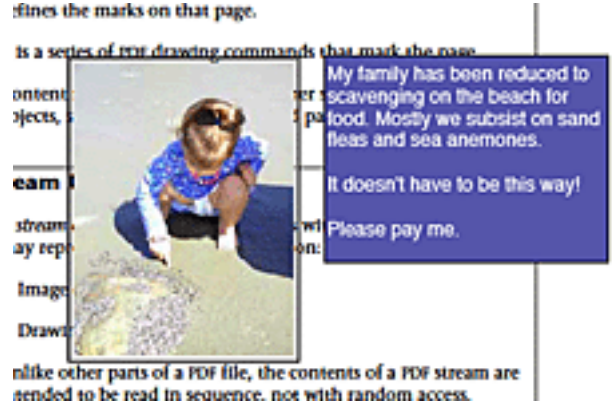
Next Page ->

Acumen Journal: Acrobat User 4

New Nags

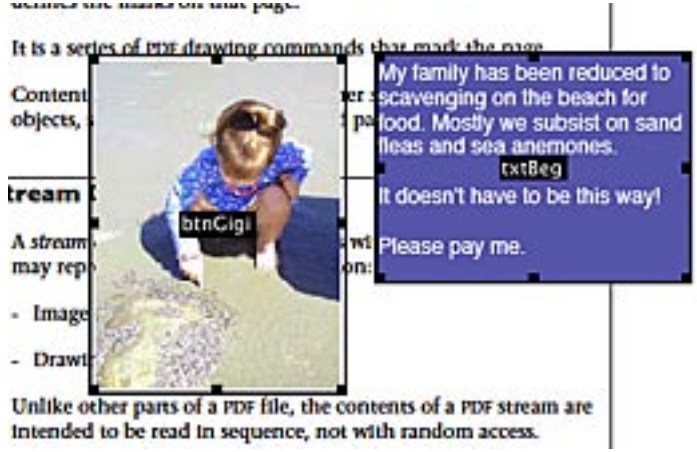
Nag 6: Add Picture

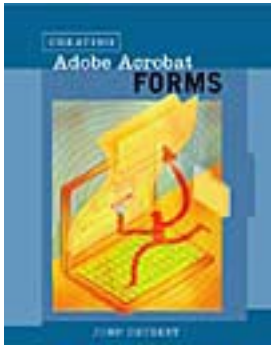
Our first new nag, invoked upon the sixth opening of the document, will display a picture and a text box pointing out the human consequences of not paying the shareware fee.



These two items are actually form fields, both initially invisible:

- A Button field named "btnGigi" that has the photograph as its label.
- A Text field named "txtBeg" whose content is set to our plea. Note that I have selected a background color for this field that improves visibility.





I refer you to either my book *Creating Acrobat Forms* or to the Acrobat on-line help to see how to create and place these form fields.

Next Page ->

The JavaScript

The JavaScript code for the new nag adds an additional *Case* clause that simply makes these fields visible.

```
switch (global.nagCount)     {    // Examine global.nagCount
    case 4:                         // If it's 4, do the following:
        ...                  // Add annotation at bottom of page
        break

    case 5:                         // If it's 5, do the following:
        ...                  // Add annotation in middle of page
        break

    case 6:      // If it's 6, get references to the form fields...
        var gigi = this.getField("btnGigi")
        var beg = this.getField("txtBeg")
                 // ...and make them visible (that is, not hidden)
        gigi.hidden = false
        beg.hidden = false
        break
}
```

Sample File

This version of the Nagware document is in this month's *zip* file with the name *Nag6.pdf*.

Incidentally, the final example in the previous article was *Nag3a.pdf*. There is no Nag4 or Nag5; I decided to change the file numbering.

Our *case 6:* clause is very straightforward; it simply sets the *hidden* property of our two form fields to *false,* making those fields visible. You may want to check the *Extending Acrobat Forms* book for the details on how this code works.

Next Page ->

Acumen Journal: Acrobat User 6

Nag 7: Make Unreadable

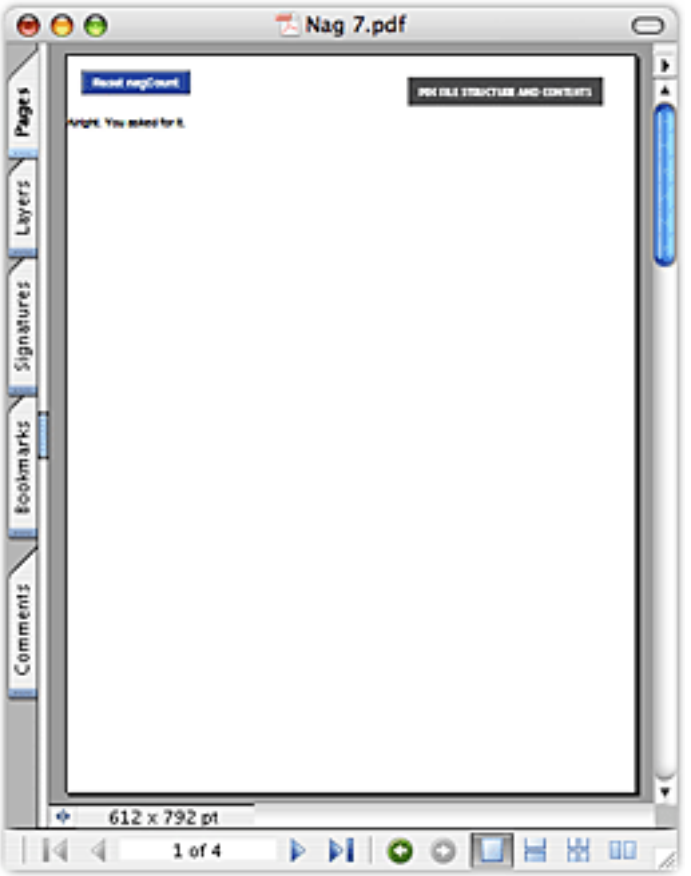
Our final nag renders the document unuseable by "whiting out" the pages. We shall do this by placing an annotation onto the page that covers the entire page. We'll do this with a call to the Doc object's *addAnnot* method, as in some of our previous nags.

Sample File

This version of the Nagware document is in this month's *zip* file with the name *Nag7.pdf*.

We are going to make this nag code the *default* clause in our *switch* statement:

```
switch(global.nagCount) {
    ...
    case 5:
        ...
        break
    case 6:
        ...
        break
    default:
        ... mask the page
}
```



Next Page ->

Remember that the *switch* statement executes the *default* clause if none of the other cases are true. In our case, if *global.nagCount* does not have any values *0* through *6,* we'll execute the *default* code, masking the Acrobat page with an annotation.

For this to work, we must supply additional *case* statements, handling *nagCount* values *0* through *3*. Since we don't want to actually do anything for these cases, we can simply have them execute a *break,* exiting the *switch* statement.

Diagramatically, our *switch* statement now looks like this:

```
switch(global.nagCount) {
    case 0:                 // Do nothing for cases 0 through 3
    case 1:
    case 2:
    case 3:
        break
    case 4:
        ... add the 1st annotation
    case 5:
        ... add the 2nd annotation
    case 6:
        ... make the picture and text field visible
    default:
        ... mask the page
}
```

Next Page ->

Acumen Journal: Acrobat User 8

To mask the pages, we use code nearly identical to what we have done before: we make a call to the *Doc* object's *addAnnot* method.

```
default:                // Time to pull off the kid gloves
    this.addAnnot({ // This annotation covers the entire page...
        page: 0,    // ...except for the top inch or so.
        type: "FreeText",
        rect: [ 0, 0, 612, 730 ],
        alignment: 1,
        contents: "Alright. You asked for it.",
        fillColor: color.white,
        textSize: 30,
        width: 0,
        readOnly: true
    })
```

This method call places a white annotation over the entire page, minus a small margin left visible at the top. (I wanted the "reset nagCount" button to be accessible.)

Pretty easy, but for one thing: this call to *addAnnot* places a mask over only the first page of the Acrobat file (the *page* property is set to *0)*. We, of course, want to place a mask over all of the document's pages. We need to have our *default* class call *addAnnot* for each page in the document. I, for one, do not want to type in the *addAnnot* call for each page in a 600-page tome.

Happily, there is a much easier way to do this, using something called a *for* loop.

Next Page ->

Acumen Journal: Acrobat User 9

Introduction to Loops

Loops are an extremely important tool in any programming language. They allow you to repeatedly execute blocks of code until some exit condition is satisfied. Here we are going to introduce the *for* loop, which will allow us to conveniently place a mask on every page in our document. If you already know how *for* loops work, you may want to skip this section and go directly to the new version of our nag code.

The *for* Loop

The *for* loop repeatedly executes a specified block of code while a particular condition is true. Generically, the loop looks like this:

```
for (initialization; repeat-while-true; increment) {
    repeated block
}
```

For example, the loop below steps through each page in the current Acrobat document and removes all the links on each page. (I'll let you look up the Doc object's *removeLinks* method in the *Acrobat JavaScript Object Reference.)*

```
for (var i = 0; i < this.numPages; i++) {
    this.removeLinks(i, [0, 0, 612, 792])
}
```

Next Page ->

for Loop Arguments

Looking at the sample on the previous page,the braces enclose the JavaScript code that is to be repeated each time through the loop. The parentheses immediately following the *for* keyword contains a set of three JavaScript code snippets separated by semicolons:

```
for (initialization; repeat-while-true; increment) {
    repeated block
}
```

- *initialization* - This is code that will be executed at the beginning of the loop. It should initialize any variables you want to use in the course of the loop. In our case, we initialized a variable *i* to *0,* the page number of the first page in a PDF file.
- *repeat-while-true* - This is JavaScript code that tests to see if some condition is true. After each time through the loop, *for* will check the value of this test and repeat the loop again if the test is true. In our example, we look to see if *i* is less than *this.numPages;* the *for* loop will therefore continue as long as *i* is less than the number of pages in our document. (Remember PDF page numbers start at *0.)*
- *increment* - This code is executed at the end of each pass through the loop. This code should increment or otherwise modify the values of any variables used within the loop. In our example, we increment *i,* moving on to the next page number.

Note that our repeated code, in the braces, uses *i* as the page number in the call to *this.removeLinks;* as we proceed through the loop, we shall look at each page in turn.

Next Page ->

Acumen Journal: Acrobat User 11

Masking All the Pages

What we want to do in our final nag is to place a white mask over all of the pages. To do this, we shall simply place our earlier call to *this.addAnnot* in a *for* loop and use our loop counter, *i*, as the page number in *addAnnot:*

```
default:
    for (var i = 0; i < this.numPages; i++){
        this.addAnnot({
            page: i,                    // Use i for the page number
            type: "FreeText",
            rect: [ 0, 0, 612, 730 ],
            alignment: 1,
            contents: "Alright. You asked for it.",
            fillColor: color.white,
            textSize: 30,
            width: 0,
            readOnly: true
        })
    }
```

Sample File

This version of the Nagware document is in this month's *zip* file with the name *Nag7a.pdf*.

Each time through the loop, *i* takes on the value of a new page number and we add a masking annotation to a new page. When we exit from the loop, we will have placed a mask over all of the pages in the document.

Next Page ->

Acumen Journal: Acrobat User 12

Registration

At this point, the nagging part of our task is done; our Acrobat document will repeatedly ask for money and eventually become unreadable if the user doesn't pay up.

Sample File

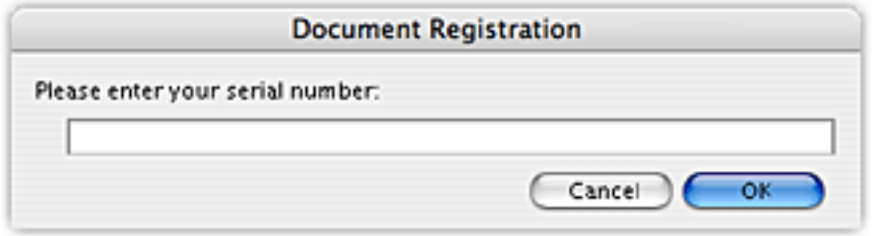
This final version of the Nagware document is in this month's *zip* file with the name *NagFinal.pdf*.

So what if the user pays up? How do we tell our document to stop nagging once the user has registered?

What we shall do use a *global.nagCount* value of *–1* to indicate the user has registered the document. We shall modify our document JavaScript so that it does nothing if *nagCount* is *-1*.

The final version of our Acrobat file, entitled *NagFinal.pdf,* adds a *Register* button. This button executes a JavaScript that does the following:

- Display a dialog box asking the user for a serial number.
- Decides whether the serial number entered by the user is valid.
- If so, then the script sets *global.nagCount* to *-1*.



It would probably be good if the script also hid the *Register* button if the serial number is valid, since the document has now been registered. However, I shall leave that as an Exercise For the Student.

Next Page ->

The Button Script

The *Register* button's Mouse Up action will execute the following JavaScript:

Button Script

This button script is in this month's *zip* file with the name *BtnScript.js*. This script also resides in the *btnRegister* Mouse Up action, of course.

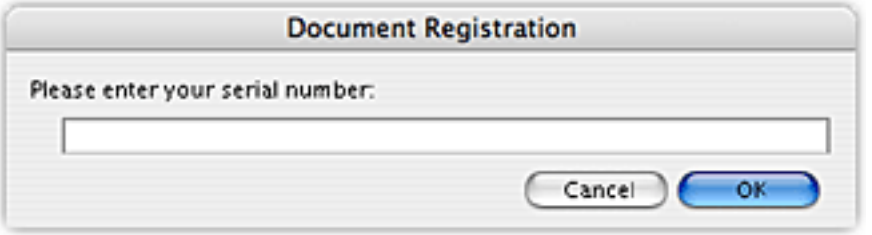
```
var s = app.response("Please enter your serial number:",
                                        "Document Registration")
if (s != null)     {
     if (s == "Password")    {    // Valid serial number?
         global.nagCount = -1    // Yes: set nagCount to -1
         app.alert("You will need to re-open this document
                                  to see its contents again.")
     }
     else                              // Not valid: tell the user
         app.alert("Sorry. That's not a valid password.")
}
```

Step by Step

Let's look at this script in detail:

```
var s = app.response("Please enter your serial number:",
                                        "Document Registration")
```

The *app* object's *response* method presents to the user a dialog box that asks for a text response. The user can type text into the text field and click *OK* or *Cancel*.



Next Page ->>

The *App.response* method takes two strings as its arguments:

- The prompt string that should be presented to the user.
- A title for the dialog box.

There are also a couple of additional, optional arguments you may pass to the method, but they have no bearing on our present script and we shall ignore them.

The method returns either the text entered by the user or *null* if the user clicked *Cancel*. In our case, we assign the text or the null to the variable *s*.

```
if (s != null)     {
     ...
     ...
}
```

If *s* is *null,* then the user clicked on the *Cancel* button and we need to do nothing. Our *if* statement will execute the JavaScript in braces if *s* is not *null*. (Remember that "!=" means "is not equal to.")

Next Page ->>

```
if (s == "Password")     {    // Valid serial number?
     global.nagCount = -1     // Yes: set nagCount to -1
     app.alert("You will need to re-open this document
                                to see its contents again.")
}
```

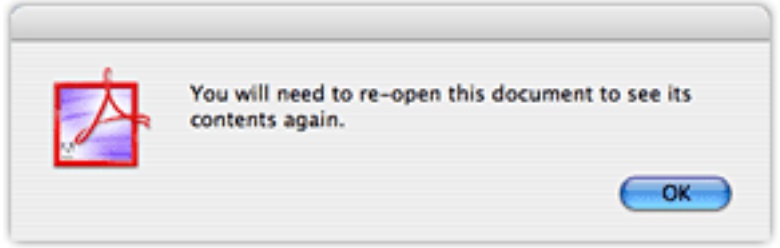
Within our *if* clause, executed if *s* isn't null, we shall examine the value of *s* to see if it is a valid serial number. In real life, this would probably entail some algorithmic calculation to see if *s* matches a pattern consistent with however you generated your serial numbers. In our case, we shall be lazy and simply see if *s* is the string "Password."

Removing the Nags

An actual *Register* button would probably want to remove any existing nags. The code to do that is beyond what I want to cover in this article.

However, if you're curious how to do this, the *Reset numPages* button removes all of the existing annotations and hides the form fields again. This button's Mouse Up script is in the zip file as *resetNagCount.js*.

If *s* is "Password", then we shall set *global.negCount* to *-1*, our signal that the user has registered. We shall also point out to the user that the reminders littering the Acrobat page will not go away until they close and then reopen the document. (We could automatically remove the nags if we wished; see the sidebar.)




```
else                              // Not valid: tell the user
     app.alert("Sorry. That's not a valid password.")
```

If *s* is not "Password", then we'll tell the user about it.

Next Page ->

Change the Document Script

We now have a button that registers the document. We have yet one more task: we must change our document JavaScript so that it treats *-1* as meaning "registered."

There are two changes we need to make to the script.

We must change our initial *if…else* clause so that we increment *nagCount* only if its value is not *-1*. A *nagCount* of *-1* will remain so indefinitely:

```
if (global.nagCount == null) {
     global.nagCount = 0
     global.setPersistent("nagCount", true)
}
else {
     if (global.nagCount != -1)
          global.nagCount++
}
```

We also need to add *-1* to the values that should be ignored by our *switch* statement:

```
switch (global.nagCount)     {
     case -1:
     case 0:
     case 1:
          break
```

Next Page ->

Acumen Journal: Acrobat User 17

The Final Script

So here is the final document script for our nagware document:

Sample File

This JavaScript is in this month's *zip* file with the name *NagFinal.js*.

```
if (global.nagCount == null)     {    // Does nagCount not exist?
    global.nagCount = 0               // Doesn't: create it
    global.setPersistent("nagCount", true)
}
else    {                             // Does:
    if (global.nagCount != -1)        // Not registered?
        global.nagCount++             // Nope: increment nagCount
}

if (global.nagCount > 2)              // Opened more than twice?
    app.alert("You haven't paid for this document.\n
                                          Please pay me.")

switch (global.nagCount)     {    // Examine nagCount's value
    case -1:                      // Ignore these values
    case 0:
    case 1:
    case 2:
    case 3:
        break
```

Next Page ->

Acumen Journal: Acrobat User 18

```
case 4:                 // First nag: discreet annotation
    this.addAnnot({
        page: 0,
        type: "FreeText",
        rect: [ 206, 24, 406, 48 ],
        contents: " Slacker! I need my money!",
        fillColor: color.red,
        strokeColor: color.transparent,
        textSize: 10,
        width: 0,
        readOnly: true
    })
    break

case 5:                 // 2nd nag: obtrusive annotation
    this.addAnnot({
        page: 0,
        type: "FreeText",
        rect: [ 206, 500, 406, 550 ],
        contents: "I mean it! I need it now!",
        fillColor: color.red,
        textSize: 30,
        width: 0,
        readOnly: true
    })
```

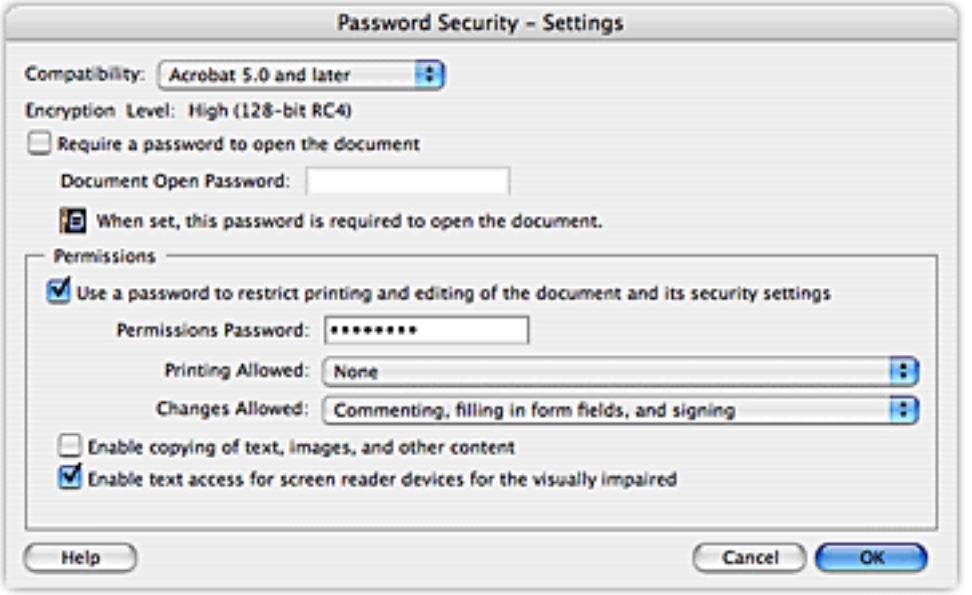
Next Page ->>

```
            break

        case 6:                 // 3rd nag: Show picture and beg
            var gigi = this.getField("btnGigi")
            var beg = this.getField("txtBeg")

            gigi.hidden = false
            beg.hidden = false
            break

        default:                // Final nag: mask the pages.
            for (i = 0; i < this.numPages; i++) {
                this.addAnnot({
                    page: i,
                    type: "FreeText",
                    rect: [ 0, 0, 612, 730 ],
                    alignment: 1,
                    contents: "Alright. You asked for it.",
                    fillColor: color.white,
                    textSize: 30,
                    width: 0,
                    readOnly: true
                })
            }
}
```

Next Page ->

Acumen Journal: Acrobat User 20

Final Notes



Lock Your Doc

Don't forget to lock your nagware document. Its security settings should prevent a user from examining (or changing) your JavaScripts.

Limitations

Keep in mind that this method of nagging does not constitute complete security for the document. Someone who is sufficiently determined and has some programming skill can eventually remove the nags. The scripts' intent is to make it unavoidably clear that you expect to be paid for your work.

In that goal, this technique is entirely successful.

Return to Main Menu

Using EPS Files in Handwritten PostScript Code

There are many organizations that write their own PostScript code, rather than using a printer driver. This is particularly common among companies that do variable data printing.

Often, the PostScript document being created needs to contain artwork created in an application such as *Adobe Illustrator* or *Photoshop*. This can lead to some headscratching, the developers often wondering if they are going to need to implement, say, a TIFF interpreter in PostScript.

The easiest way to support externally-created graphics is to export them from the design application as Encapsulated PostScript; EPS files, as we shall see, are extremely easy to embed in your own PostScript code. If you wish, and circumstances permit, you can leave the EPS file in the original file and just embed a reference to that file into your PostScript code.

This month, we shall discuss how to incorporate an EPS file as an illustration in your own, handwritten PostScript code.

Next Page ->>

Review: EPS File Format

We discussed EPS files in full detail in your PostScript class; you may want to review your student notes for the full discussion.

As a brief reminder (and introduction for people who have unaccountably not taken a PostScript class), let's review the contents and structure of EPS files.

PostScript + Preview

An Encapsulated PostScript file consists of two parts:

- The *PostScript* that should be used to produce the illustration at print time
- An optional *preview graphic* that is intended for the use of the importing application.

When you import an EPS file into *Adobe InDesign,* say, that program displays the preview on the screen, giving you something you can drag around and resize as needed for the layout.

At print time, InDesign does the following:

- Works its way through the document, generating its own PostScript code
- When it encounters the imported EPS file, InDesign opens the EPS file, grabs the PostScript code it contains, and sends that code to the printer. It preceeds the code with a *translate* and *scale* that moves and resizes the EPS graphic as needed by the layout.
- InDesign then resumes creating its own PostScript code until the next EPS file.

Next Page ->

Acumen Journal: PostScript Tech 23

PostScript Requirements

The PostScript code in an EPS file is just regular PostScript, subject to three requirements:

1. The first line must be the following (though the numbers can be different):

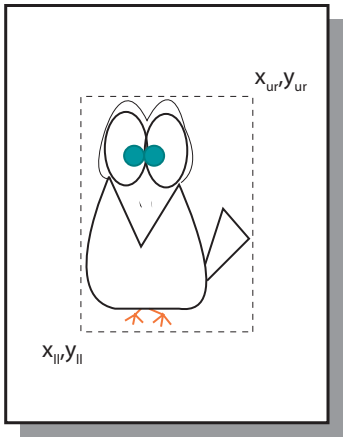
   ```
   %!PS-Adobe-3.0 EPSF-3.0
   ```

   This indicates two things about this file: it's a PostScript file adhering to the Document Structuring Convention version 3.0 (or whatever); furthermore, it's an *encapsulated* PostScript file adhering to EPS spec version 3.0 (or whatever).

2. Somewhere among the comments at the beginning of the file, there must be a BoundingBox comment:

   ```
   %%BoundingBox: x_ll y_ll x_ur y_ur
   ```

   The *BoundingBox* keyword is followed by four numbers, the User Space *x* and *y* coordinates of the lower-left and upper-right corners of the file's *bounding box* (the rectangle that exactly encloses the EPS graphic). This tells the importing application where the EPS file prints on the page.



3. Finally, the PostScript code must be "well behaved," that is, it must do nothing that is inconsistent with being used as an illustration; it must not erase the page, reinitialize the graphics state, or do anything else that would make it difficult to import.

Next Page ->

Acumen Journal: PostScript Tech 24

EPS File Format: Microsoft Windows

In Microsoft Windows, the EPS file on your disk may have two possible formats:

No Preview

If there is no preview (remember, it's optional), then the EPS file will be just a PostScript file that starts with *%!PS-Adobe-*… etc.

Preview Present

If there is a preview, then the EPS file will have three sections:

- A 30-byte header containing directory information (check your student notes for the details).
- The PostScript code
- The preview as either a TIFF image or Windows metafile graphic.

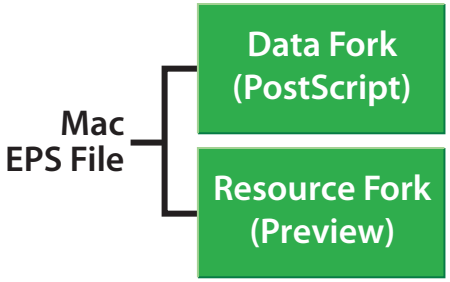
The PostScript and preview may be in either order.

Next Page ->

File Format: Macintosh

The Macintosh has a unique file system. Every Macintosh file can have two separately-addressable parts:

- A *Data Fork,* which corresponds to what would be the file on other systems.
- A *Resource Fork,* which contains a collection of programmer-defined bits of data, retrievable by type, number, and (optionally) name.



One of Apple's challenges in designing OS X was figuring out how to reconcile this quite powerful, but idiosyncratic, file system with the UNIX underpinnings of the new operating system.

Macintosh EPS files store the PostScript code in their data forks; the preview, if any, is stored in the resource fork (as PICT resource #256, if you were curious).

Next Page ->

File Format: All Others

EPS files made on any system other than Mac or Windows (UNIX, for example) have *EPSI* previews ("Encapsulated PostScript Screen Image"). An EPSI preview is simply a screen image embedded in-line with the PostScript code as a series of comments.

The PostScript header (the series of comments at the beginning of the file) is followed by a *%%BeginPreview* comment, which is followed, in turn, by hexadecimal image data:

```
%!PS-Adobe-3.0 EPSF-3.0
%%Creator: GIMP PostScript file
...
%%BoundingBox: 14 14 521 681
%%EndComments
%%BeginPreview: 255 256 1 256
% aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
% ffffffffffffffffffffffffffffffffffffffffffffffff
% aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
% ffffffffffffffffffffffffffffffffffffffffffffffff
...
%%EndPreview
```

Note that each line of image data begins with a percent sign so that a PostScript interpreter will treat it as a comment.

See your student notes for the meaning of the numbers after the *BeginPreview* keyword.

Next Page ->

Acumen Journal: PostScript Tech 27

Using the EPS File

To use an EPS file as a graphic in your own PostScript output, you need simply take the PostScript code from the EPS file and place it in-line with your own PostScript:

```
...
... Your PostScript Code
...
... PostScript code from EPS file
...
... Your PostScript code, again
...
```

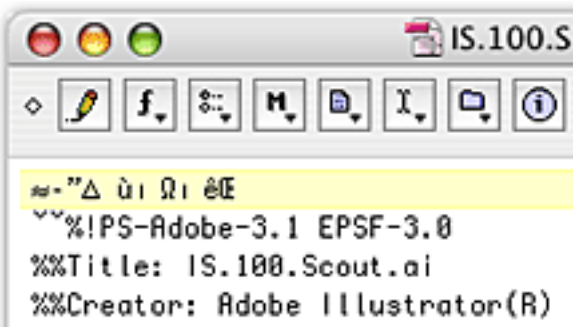
Preparing the EPS File

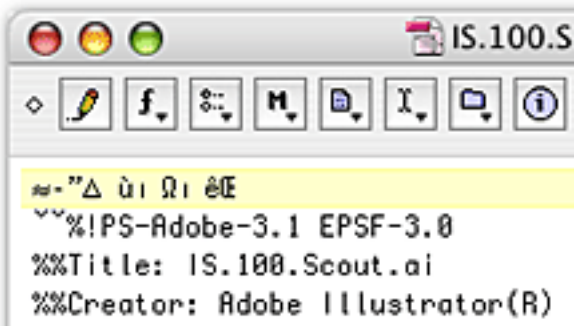
There is some initial preparation you may need to perform on the EPS file; what is necessary depends upon the system on which the EPS file was created:

Windows

If the EPS file came from a Windows system, then you will need to strip off the header and preview. Just open the EPS file in a text editor; you will see some binary garbage in front of the *%!PS-Adobe*.

Next Page ->





There might be only a small amount of binary, as at right, representing the 30-byte EPS header; on the other hand, there may be several kilobytes of binary data, consisting of both the EPS header and the preview data. In either case, erase everything up to (but not including) the "%!".

If the initial binary data was small, as in the illustration, then the preview still resides within the EPS file, following the PostScript. Scroll to the end of the PostScript within the file (look for the "%%EOF") and erase the several kilobytes of binary stuff you find there.

Now, you are left with only the EPS file's PostScript code, suitable for insertion into your own PostScript program.

Macintosh There's no preparation necessary before using a Macintosh-style EPS file. The preview is nicely isolated in the file's resource fork, which will be ignored by the text editor you use to copy and paste the PostScript code into your own program.

Next Page ->>

Everyone Else

For UNIX-style EPS files, the only preparation worth mentioning might be to remove the EPSI preview from within the PostScript code. (Why retain several kilobytes of commented preview bitmap?)

This is easily done, since the EPSI preview is bounded by well-defined markers. The preview data is preceedd by the *%%BeginPreview* comment and is terminated by the *%%EndPreview* line.

Delete everything from *%%Begin-* to *%%EndPrevew* and you will be left with only the usable PostScript code.

Next Page ->>

Placing the File

So we have now prepared our EPS file and are ready to use it in our PostScript code. We shall need to do a bit of work before and after the EPS' code in order to use it successfully.

Here's what you need to put into your PostScript code to use a EPS file:

1. A *%%BeginDocument* DSC comment
2. A call to *save*
3. Calls to *translate* and *scale* (and maybe *rotate)* to reposition and resize the EPS graphic as needed for your document.
4. Redefinitions of *showpage* and *setpagedevice* that render them harmless.
5. The EPS PostScript code.
6. A call to *restore.*
7. A *%%EndDocument* comment.

Let's talk about these in a bit of detail.

1. %%BeginDocument

This step is often omitted and, indeed, ignoring it will not prevent your EPS from working. Still, it's good programming style in PostScript to preceed embedded EPS code with a *%%BeginDocument* and follow it with a *%%EndDocument*.

Next Page ->>

These comments are, of course, part of the Document Structuring Convention, that set of rules that all professional-grade PostScript should follow.

The *BeginDocument* comment has the following form:

```
%%BeginDocument: MyIllustration.eps
```

Note that the keyword itself is followed by the name of the original EPS document.

2. Do a *save*

You are about to embed in your program a clump of PostScript code of arbitrary length and complexity. You will want to recover the VM used by this code and erase whatever key-value pairs it may define. You do this, of course, by calling *save* before the EPS code and *restore* afterward.

As you know, *save* returns a saveobject; we shall store this return value as a key-value pair for easy recovery later:

```
/preEPSState save def
```

Of course, the name "preEPSState," above, is arbitrary; pick something descriptive that you like.

Next Page ->>

3. Position and Resize the EPS

The graphic produced by an EPS file, if left to itself, occupies a position on the page described by the file's *BoundingBox* comment:

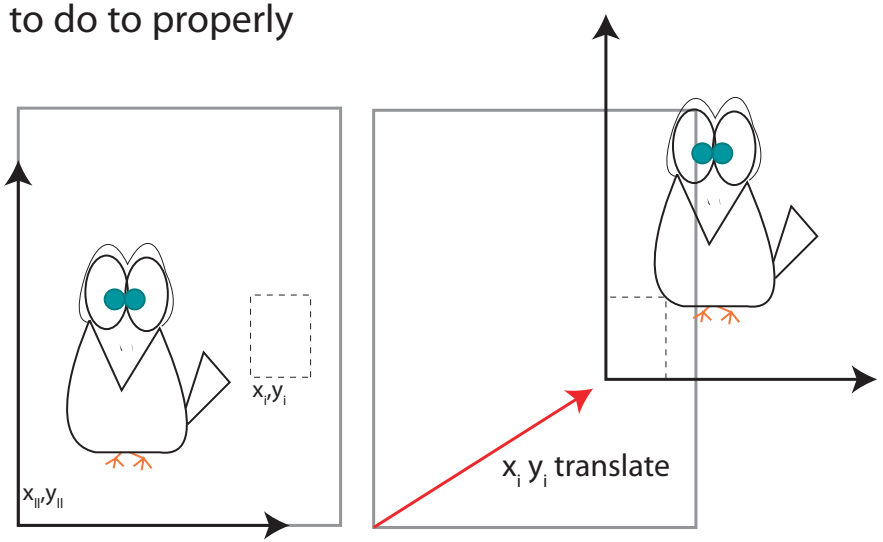
```
%%BoundingBox: x_ll y_ll x_ur y_ur
```

You will need to precede the EPS file's PostScript code with a set of calls to *translate* and *scale* that moves the EPS graphic to the position and size you want for the final illustration. This is a three step process (although one step may sometimes be omitted).

Presume that you want the lower left corner of the final illustration to be located at x_i, y_i. Here's what you would need to do to properly place the EPS image:

1. Translate by $x_i\ y_i$. This moves the origin of the EPS' PostScript code to the position you want for the illustration.

   Note that this does *not* necessarily place the graphic, itself, where you want it; the EPS file may not draw its contents at *0,0*.

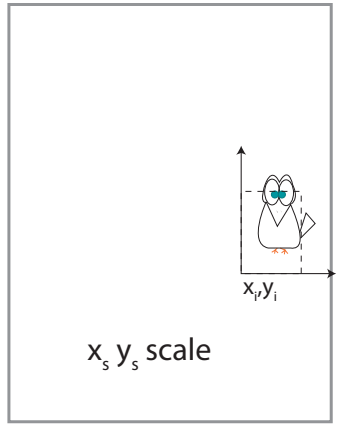


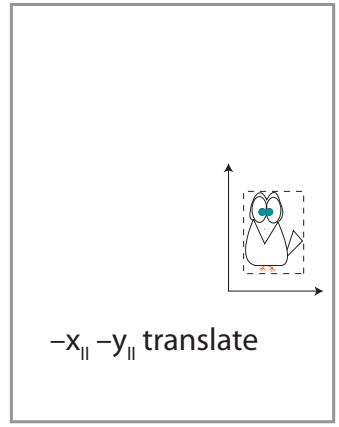
Next Page ->

2. Scale by whatever amount is needed (call it s_x s_y) to change the size of EPS graphic to what you need for the illustration.

3. Translate by $-x_{ll}$ $-y_{ll}$. This moves the EPS graphic to the postiion you want for the illustration.

   This third step can be omitted if the EPS file draws its graphic at the origin, that is, if the file's x_{ll} and y_{ll} are both *0*.





4. Redefine Operators

There are two PostScript operators that a proper EPS file should never call, but that do, nonetheless, occur in many Encapsulated PostScript files. The *showpage* and *setpagedevice* operators are very inappropriate within an EPS file and will make the file unuseable. A *showpage* in the EPS file would cause a premature page ejection; *setpagedevice* will erase the entire page.

You must therefore defang the EPS file by redefining these operators to something harmless:

```
/showpage { } def
/setpagedevice /pop load def
```

These two lines redefine *showpage* to do nothing and *setpagedevice* to simply discard its dictionary argument. You will get the original definitions back when you execute the eventual *restore*.

Next Page ->

You may wish to consult your level of personal paranoia to decide if there's anything else you want to redefine. I've seen PostScript output that redefines *copypage, setdash, and setlinecaps* for example.

5. Embed the PostScript

Now you can place the PostScript code taken from the EPS file into your program.

6. Do a *restore*

After the EPS code, execute *restore,* using the saveobject from your earlier call to *save:*

```
preEPSState restore
```

This will reclaim the memory used by the EPS file's code, discard any key-value pairs defined in the EPS file, and cause *showpage* and *setpagedevice* to revert to their original definitions.

7. *%%EndDocument*

You should finish up the whole embedded EPS block with an *EndDocument* comment:

```
%%EndDocument
```

Again, nothing will break if you don't do this, but your friends will admire you greatly if you include it.

Next Page ->

Acumen Journal: PostScript Tech 35

All together

So here is what the EPS block should look like in your PostScript program:

```
%%BeginDocument: MousyToes.eps
/preEPSState save def                % Initial save
xi yi translate                      % Reposition and resize
sx sy scale
-xll -yll translate
/showpage { } def                    % Defang
/setpagedevice /pop load def
...
... EPS Code goes here
...
preEPSState restore                  % Reclaim memory; erase defs
%%EndDocument
```

That's all there is to it. Easy, isn't it?

Final Notes

EPS Files and Images

In my *Variable Data PostScript* class, one of the most common questions people ask at the beginning is "Are we going to see how to include TIFF files in our PostScript code? Please?"

Of course, they usually don't care about TIFF, *per se;* they just need to embed images supplied by their clients in the variable data documents they produce for their clients.

Next Page ->

Acumen Journal: PostScript Tech 36

PostScript has no direct support for TIFF, BMP, PNG, or any of the other common image formats. But that's alright; all image editing software will export your images to EPS and then you can then use that image with the technique described in this article.

Externally-Stored EPS

It may be convenient for you to leave the EPS PostScript code in an external file and then simply execute it from within your code using *run*.

I refer you to the January 2002 *Journal* to see how to save PostScript code to your printer's hard disk. Having done so, instead of including the entire EPS PostScript stream, you can simply execute the contents of the external file with a *run:*

```
(MyExternalFile.ps) run
```

This is especially useful if this EPS file is used many times within the PostScript stream, as is common in variable data jobs.

EPS: *A Force For Good*

If you do variable data or any other handwritten PostScript code, EPS is the greatest thing since coffee. It allows you to effortlessly incorporate into your PostScript output graphics and images from any professional software.

Return to Main Menu

Schedule of Classes, Dec 2004 - Mar 2005

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			Feb 21–24
PostScript Foundations		Jan 31–Feb 4	
Variable Data PostScript			
Advanced PostScript			Mar 7–11
PostScript for Support Engineers	Dec 6–10		Mar 21–25
Jaws Development		*On-site only*	

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

Registration Info

Acrobat Class Schedule

These classes are taught occasionally 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

Acrobat Class Fees — *Acrobat Essentials* and *Creating Acrobat Forms* (½-day each) cost $180.00 or $340.00 for both classes. 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: 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

Back 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?

New PDF Class

Nothing too new this month. I am still in the early stages of laying out the second PDF File Content and Structure class. The topic list is below; as before, if you think something should be added to or dropped from this list, send an email to john@acumentraining.com.

Preliminary Topic List

- Overprinting
- CID Fonts
- Halftones
- Marked Content
- Rendering Intents
- Smooth shading
- Reference XObjects
- Cross reference streams
- BX & EX
- File Spec
- Masked Images
- Digital Signatures
- AcroForm
- Transfer Functions
- Shape dictionaries
- Layers
- Name Dictionaries
- Patterns
- Composite Fonts
- Linearized PDF
- Stroke Adjustment
- Halftones
- Text Knockout
- Object streams
- More on data structures

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 want to move to Canada?

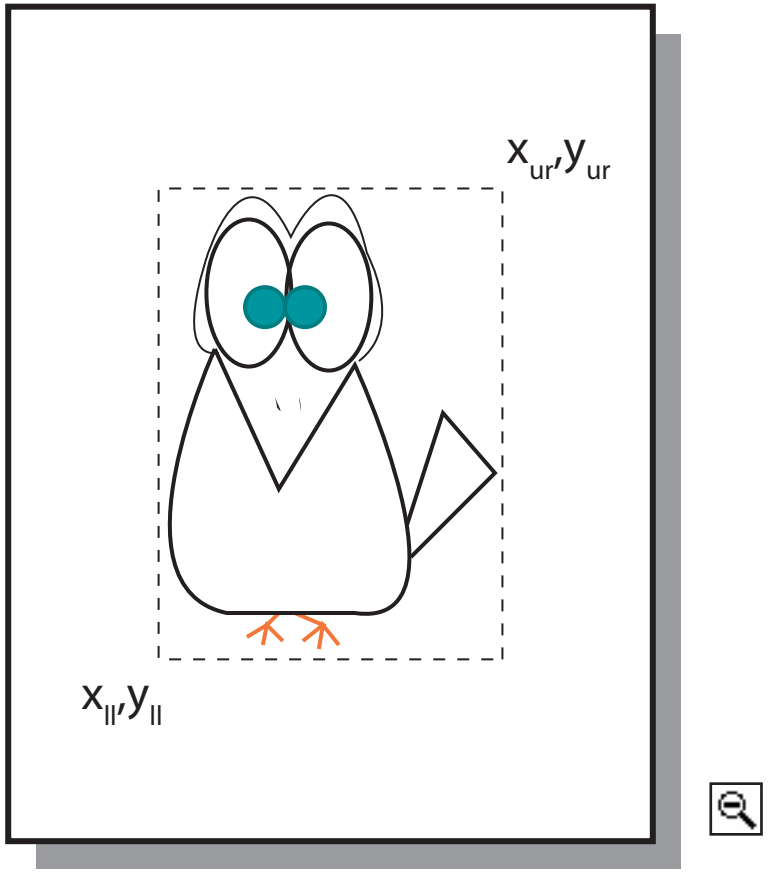
Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

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


x_{ur}, y_{ur}
x_{ll}, y_{ll}

efines the marks on that page.

is a series of PDF drawing commands that mark the page

ontent
ojects,

eam

stream
ay rep

Image

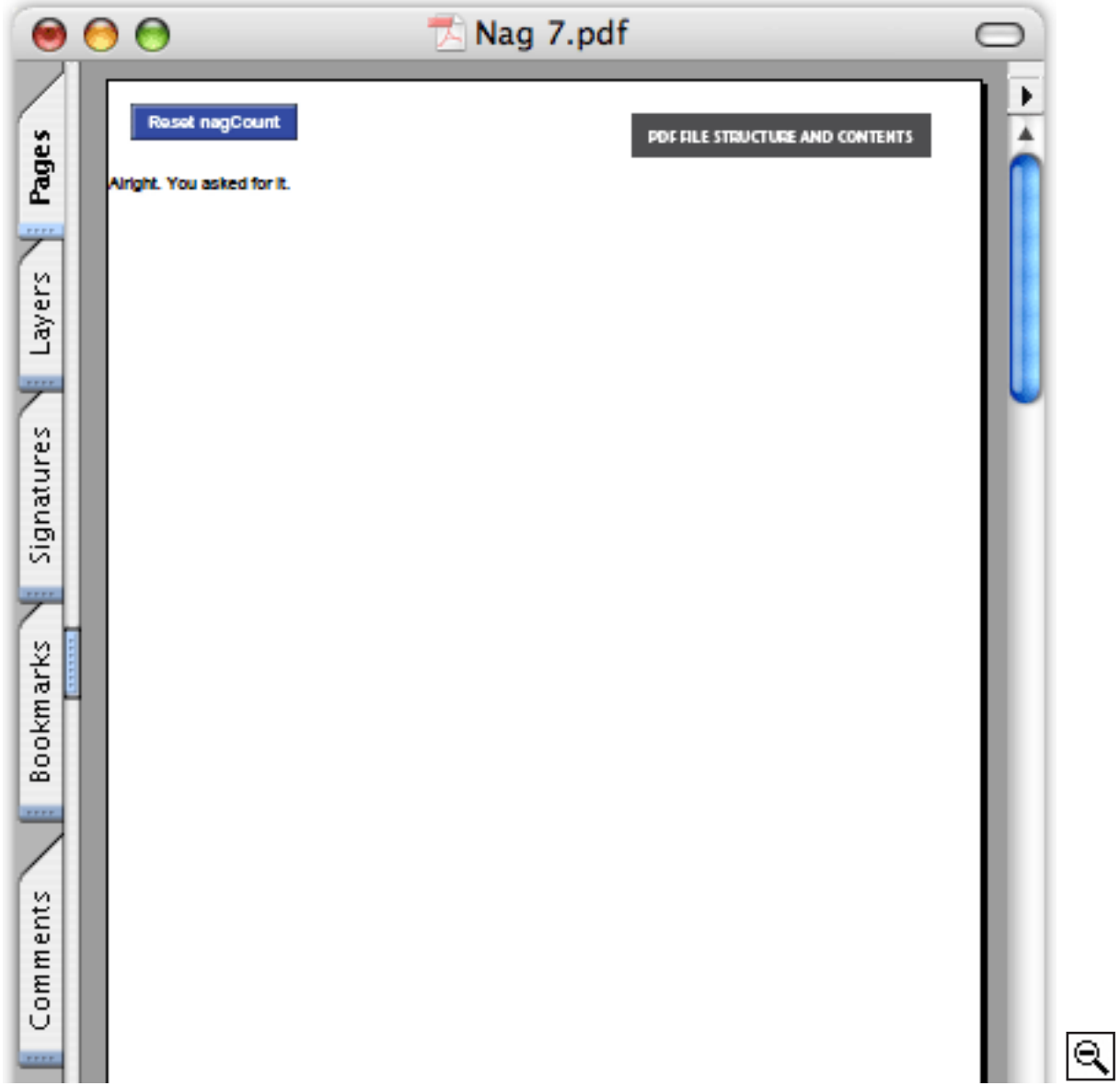
Drawi

My family has been reduced to scavenging on the beach for food. Mostly we subsist on sand fleas and sea anemones.

It doesn't have to be this way!

Please pay me.

nlike other parts of a PDF file, the contents of a PDF stream are
tended to be read in sequence, not with random access.


Nag 7.pdf
Pages
Layers
Signatures
Bookmarks
Comments
Reset nagCount
PDF FILE STRUCTURE AND CONTENTS
Alright. You asked for it.