# Table of Contents

Acumen Training

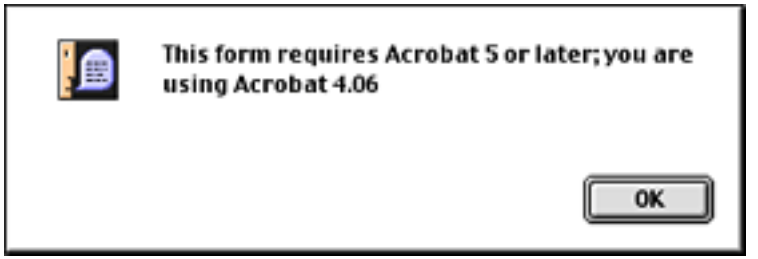
# Checking the Acrobat Version

Each version of Acrobat has added much to the capabilities of the Acrobat package. Consider, for example, support for electronic forms. Acrobat 3 had only very simple support for form fields. With Acrobat 4, forms came into their own, adding support for a very rich array of form field types. Acrobat 5 provides a very much richer JavaScript interface and some additional event types to which you may attach actions in a form.

If you design an Acrobat document that makes use of more-current features, it is important that the people reading your Acrobat file be using a sufficiently-recent version of Acrobat or the Reader.

How do you, as the document designer, ensure that the user has a version of Acrobat that will correctly handle your PDF file?

The answer: you set up your Acrobat file so that when a user opens the file, a JavaScript program checks the user's version of Acrobat Viewer and puts up a warning if they have too primitive a version.




We shall do this by associating the JavaScript with the *Page Open* event for our document's first page. Acrobat will execute this JavaScript program every time it opens the PDF file.

Let's see how to do this.

Next Page ->

## Creating the *Page Open* Action

A *Page Open* action is an activity that is carried out whenever Acrobat opens a particular page in a PDF file. We are going to use this mechanism, associating a version-checking JavaScript with the first page of our Acrobat file.

Start within Acrobat by going to the first page in your document. Then do the following:

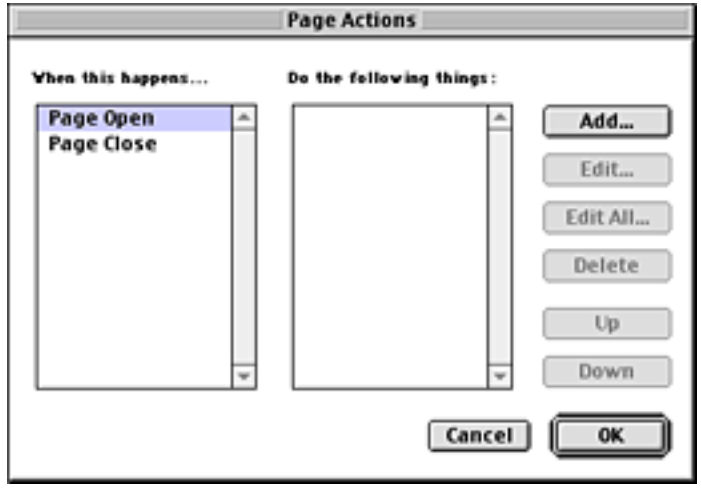
*1. Select "Set Page Action…"*

This is located in Acrobat's *Document* menu.


You will now be looking at the *Page Actions* dialog box. This dialog box is similar to the *Actions* panel in the Form Fields Properties dialog box. Here you may associate one or more Acrobat Actions with either the opening or closing of this page.

*2. Select the "Page Open" event and Click "Add"*

We want to associate a JavaScript with the *Page Open* event, to be carried out when Acrobat opens the page; select *Page Open* in the *When this happens…* list.




Click on the *Add…* button to get to the "Add an Action" dialog box (next page). This lets you specify an action to be associated with *Page Open.*

Next Page ->

*3. Select "JavaScript" from the "Type" Menu*

The *Type* pop-up menu specifies which action you are going to associate with the *Page Open* event. Select *JavaScript.*

Notice, in passing, that there are a *lot* of other actions you can associate with *Page Open:* play movies, sounds, etc. We'll talk about those sometime.


*4. Click the Edit… button*

When you click on the *Edit…* button, Acrobat presents you with a very simple text editing field into which you may type a JavaScript program.

Next Page ->

*5. Type in the JavaScript*

The JavaScript at right and below returns an alert if the user's version of Acrobat is less than 5. This is exactly what you would use if your Acrobat document uses form fields, events, or other features that were introduced in Acrobat 5. (We'll describe this JavaScript in more detail shortly.)


```
var v = app.viewerVersion;

if (v < 5)
     app.alert("This form requires Acrobat 5 or later; you are using
     Acrobat " + v);
```

Note that the "app.alert" line should be one line of JavaScript, as in the illustration above.

If you want to check for Acrobat version 4 or later, simply replace both instances of the numeral "5" with "4." (All forms should check for at least Acrobat 4, since Acrobat 3's form support was minimal.)

Click the *OK* button when you are finished.

Next Page ->

*Back out of the dialog boxes*

At this point, you should be once again looking at the Add an Action dialog box. Click *Set Action* to return to the *Page Actions* dialog box.

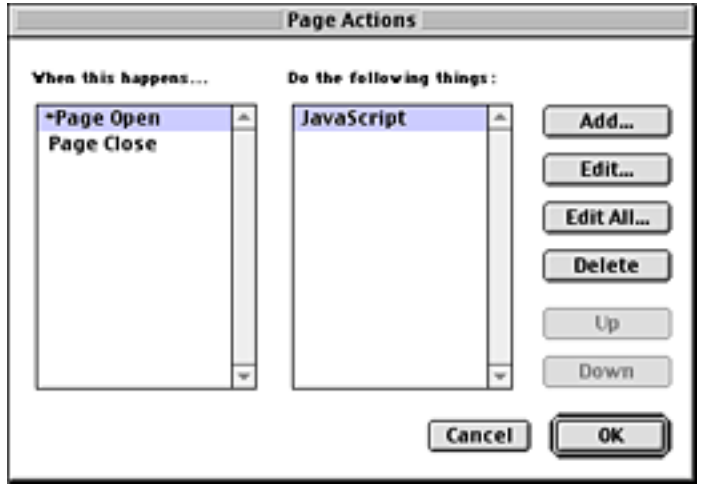
Then click *OK* in the *Page Actions* dialog box to return to your Acrobat page.


*Finished!*

Now, every time anyone turns to this page in the PDF document, Acrobat will run the JavaScript and your reader will get an alert if he or she does not have the minimum version of Acrobat.

Save the PDF file and you're done.

Next Page ->



## The JavaScript Explained

The JavaScript we typed in as the Page Action was:

```
var v = app.viewerVersion;

if (v < 5)
     app.alert("This form requires Acrobat 5 or later; you are using
     Acrobat " + v);
```

Let's look at each line of this script and see what it does. (The description here is a bit imprecise and assumes no programming experience.)

**`var v = app.viewerVersion;`**

The term *var* (short for "variable") creates a JavaScript named reference *(v,* in this case) to some piece of information. We shall use *v* to refer to the version number of the software being used to display the PDF file.

The term *app* refers to the application being used to view the Acrobat file (be it Acrobat, the Reader, or Acrobat Approval). The phrase "app.viewerVersion" is a request to the application to tell us its version number.

Loosely translated into English this line of JavaScript says:

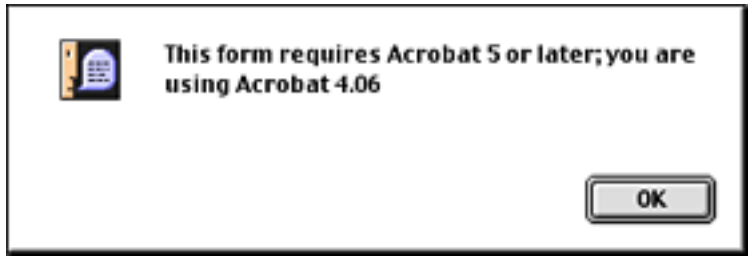
"Get from the viewer application its version number and give that number the name *v."*

Next Page ->

```
if (v < 5)
     app.alert("This form ... you are using Acrobat " + v);
```

This line starts by saying "If the version number is less than 5, then do something." What we do in this case is ask the application (again, referred to as "app") to display an alert. Whatever is in the parentheses following the verb "alert" will be displayed in the dialog box.

In this case, the alert box will have the text "This form requires Acrobat 5 or later; you are using Acrobat" followed by the current version number, as at right.




Since the *v* in the parentheses is not inside the quotes, the alert displays the value of *v* (that is, the current version number), rather than the character *v.*

Next Page ->

## Where to Learn More

JavaScript in Acrobat is stunningly useful. If you are interested in learning more about it, there are actually two steps to the process:

1. Learn the JavaScript language.

   You can do this from any number of very good books available in your local book-store. All of these describe how to use JavaScripts in a web page, rather than in Acrobat, but they will teach you how to use the language itself.

2. Learn how to use JavaScript within Acrobat.

   The only documentation currently available for this is the *Acrobat JavaScript Object Specification,* available directly from the Acrobat (not the Reader) *Help* menu. This document tells you how to manipulate Acrobat with JavaScript. This is what tells you that "app" refers to the current viewer application and that "app.viewerVersion" fetches the Acrobat version number.

   The *JavaScript Object Specification* is a reference volume, not a tutorial; it very much assumes you know JavaScript.

   Again, this document ships with the full Acrobat package; you can get to it by selecting *Help > Acrobat JavaScript Guide* (Acrobat 5) or
   *Help > Forms JavaScript Guide* (Acrobat 4).

Return to Main Menu

# Converting .pfb Files to PostScript

How do you embed a font in a PostScript file?

Windows-style downloadable font files (suffixed *.pfb,* “printer font, binary”) are tantalizingly close to being useable. But if you just send the contents of the file directly to a PostScript printer, you just get an *undefined* error; not useful.

This month, we’re going to see how to extract the font defined by a *pfb* file to a straight PostScript file. This PostScript font definition may be embedded in another PostScript program and the font will be available for use.

Next Page ->

## PFB Font Format

A downloadable PostScript font, be it Type 1 or Type 3, Mac or Windows, is really just a PostScript program. If you pop open a Windows *pfb* file in a text editor, you will be looking at PostScript code.

This code is conceptually very similar to the make-a-font exercise we do in the *PostScript Foundations* class: it makes a dictionary, transfers it to the dictionary stack, and then defines into it *FontType, FontMatrix,* and all the other things required by a PostScript font.

Note that this fragment begins with some binary data in front of the actual PostScript code.

The pfb file contains PostScript, but not *only* PostScript.

Keep reading.

```
Äô%!PS-AdobeFont-1.0: Courier-Bold 002.004
%%CreationDate: Tue Sep 17 14:02:34 1991
%%VMusage: 31992 40360
...
11 dict begin
...
/Encoding StandardEncoding def
/PaintType 0 def
/FontType 1 def
/FontMatrix [0.001 0 0 0.001 0 0] readonly def
/UniqueID 36384 def
/FontBBox{-113 -250 749 801}readonly def
```

Next Page ->

Eventually, the PostScript code executes the *eexec* operator ("Encrypted Execution"); the remainder of the file contains encrypted PostScript code.

The PostScript standard encryption was the topic of the May 2001 *Journal.* You should read that article if you are curious about the details of the encryption. (Go to the Acumen Journal page on the Acumen Training website.)

```
currentdict end
currentfile eexec
ÄËÅ–NïkxRiß/ó-ÏƒW¢>§˘Féî^ä}•tÛ*\ÄüÆ¸Øs¸ ï
...
```

It's important to understand that the encrypted part of the file (which makes up the largest part of the font definition) is just PostScript, encrypted with the standard PostScript encryption scheme. If you decrypt this data (it may be in either binary or hexadecimal format), you would find it creates a dictionary named *CharStrings* and populates that dictionary with paired character names and character definitions. (Each definition is in the form of Type 1 drawing commands in a string.)

Next Page ->

Acumen Journal: PostScript Tech 12

**File Segmentation**

The PostScript code in a *pfb* file is cut up into a series of segments. Each segment starts with a six-byte header that specifies the length of that segment and how its contents should be handled.

The format of each segment is as follows:

*Byte 0* Must be set to the value *128.* No, I don't know why.

*Byte 1* A processing instruction that tells you what to do with the contents of this segment. It will be one of the following values:

| | |
|---|---|
| 0 | The segment should be sent to the printer as-is. It contains either plain PostScript or binary data in ASCIIHex format. |
| 1 | The segment should be converted from binary to ASCIIHex before sending to the printer. |
| 2 | This value specifies end-of-file. |

*Bytes 2–5* This is a four-byte length of the segment data, low-byte-first. (Little endian is sensible here, since *pfb* is primarily used on Intel machines.)

*Bytes 6–n* This is the segment data, PostScript code encrypted or in clear.

Next Page ->

### Downloading a PFB Font

Downloading a font stored in *pfb* format is a relatively straightforward activity:

- Read the header.
- Read the next *n* bytes and send them to the printer (or target PostScript file), converting to hexadecimal if needed.
- Repeat the first two steps until you run out of file or until the processing code turns up *2.*

Next Page ->

## Abbreviated C Code

Let's look at some C code that converts a pfb file to a straight PostScript file. This example is *very* heavily edited; the full program is available among the PostScript samples on the Acumen Training Resources page. Look for the file *PFBtoPS.c.*

*Constants*

First, let's define some constants:

```
#define   kFontFileName       "COB_____.PFB"      // Source pfb file
#define   kDestFileName       "CourierBold.ps"    // Destination PS file
#define   kBigEndian          true           // Set to false if necessary


enum ErrCode {                          // Error codes returned by procedures
     kErrNoErr,
     kErrFileOpenFailed,
     kErrMemory
};

enum {                                  // Header codes indicating how to handle
     kTypeASCII = 1,                    // each section in the .pfb file.
     kTypeBinary,
     kTypeEOF
};
```

Here, we've hardwired the file names; in real life you'd want a user interface (command line, if nothing else) to specify this.

Next Page ->

*A Typedef* We'll define a typedef'd structure that will hold the header information for each segment.

```
typedef struct {
     char firstByte;
     char segmentType;
     long length;
} SegmentHeader;
```

*main()* Our *main()* routine does the following:

- Open the source and destination files.
- Loop through each segment in the file; for each segment:
  - Read the header, checking for end-of-file.
  - Send the segment to the destination file, converting to Hex, if needed.
- Close the source and destination files.

Next Page ->>

For brevity, I've removed pretty much all of the error checking in this code; the file on the Acumen Training Resources page is complete.

```
int main(void)
{
     FILE           *srcFile, *destFile;
     SegmentHeader  hdr;
     boolean        notDone = true;
     enum ErrCode   err;

     // Open the source and destination files
     srcFile = fopen(kFontFileName, "rb");
     destFile = fopen(kDestFileName, "wb");

     // Loop through all of the segments
     err = kErrNoErr;
     do   {
          notDone = GetSegmentHeader(srcFile, &hdr);
          if (notDone)
               err = SendSegment(srcFile, destFile, &hdr);
     } while (notDone && !err);

     fclose(srcFile);
     fclose(destFile);

     return err;
}
```

Next Page ->>

*Get the Header*

The *GetSegmentHeader* function reads the segment header (hence the name), returning *false* if we're at end-of-file.

```
boolean GetSegmentHeader(FILE *src, SegmentHeader *hdr)
{
     fread(&(hdr->firstByte), 1, 1, src);    // Read mystery byte 128
     fread(&(hdr->segmentType), 1, 1, src);  // Read processing code
     fread(&(hdr->length), 1, 4, src);       // Read segment length

     // The length is always low-byte-first in the .pfb file
     if (kBigEndian)
          ReverseBytes(&(hdr->length));

     // Return 'true' if the segment type indicates EOF
     return (hdr->segmentType != kTypeEOF);
}
```

You might be tempted to read the segment header all in one throw:

```
fread(hdr, 1, 6, src);
```

I was reminded (empirically) that some implementations of C pad 1-byte structure fields to word boundaries, preventing this from working.

I'll leave out the definition of *ReverseBytes;* it's boring.

Next Page ->

*Send off the Segment*

Now we write the segment data to our destination file, converting to ASCII, if necessary; again, I've left out much of the error handling.

```
enum ErrCode SendSegment(FILE *src, FILE *dest, SegmentHeader *hdr)
{
     char *srcData;

     // Allocate an appropriately-sized buffer
     srcData = (char *)malloc(hdr->length);

     // Read the segment data, converting to ASCII if needed.
     fread(srcData, 1, hdr->length, src);
     if (hdr->segmentType == kTypeBinary)    {
          srcData = ConvertToASCII(srcData, hdr->length);
          hdr->length *= 2;
     }

     // Send the data on its way.
     fwrite(srcData, 1, hdr->length, dest);
     free(srcData);
     return kErrNoErr;
}
```

The *ConvertToASCII* function converts the buffer's contents to hexadecimal "in-place." (It reallocates the buffer.)

Next Page ->

**That's all there is to it**

The file produced by this C program contains the PostScript contents of the *pfb* file. You can embed this PostScript font definition in any other PostScript file and use the resulting font. If you place

```
true 0 startjob pop
```

at the beginning of the file, the font will be installed "behind the Server Loop" and will be available to all future jobs on that printer until you turn it off.

Next Page ->

Acumen Journal: PostScript Tech 20

## Macintosh Font Files

Macintosh font files are conceptually similar to the *pfb* files we've been discussing here. The PostScript font definition is cut up into a series of segments, each identified according to how they should be handled.

If you're not familiar with the Macintosh file system, each file on the Mac has two separately-addressable pieces: the *data fork* (which corresponds to what other OS's think of as a file) and the *resource fork,* which contains a set of program-definable bits of data, each identified by a type, a number, and an optional name.

Those segments reside in the file's resource fork as a series of POST resources numbered sequentially from 501. Each POST resource begins with a one-byte code that indicates the contents of that segment:

| | |
|---|---|
| 0 | Comment; don't download to the printer. |
| 1 | Send to printer as-is |
| 2 | Convert to ASCII Hex and send to the printer |
| 3 | Send an end-of-file marker to the printer. (This is usually a control-D for serial and parallel communication.) |
| 4 | Indicates the font definition is in the data fork |
| 5 | End-of-Font. You may stop processing the font definition, though you should *not* send an end-of-file indication to the interpreter. |

I will eventually write a Mac font converter and put it among the PostScript resources on the website; check back occasionally.

Return to Main Menu

# Schedule of Classes, Sept – Nov, 2002

Following are the dates and locations of Acumen Training's upcoming PostScript and Acrobat classes. Clicking on a class name below will take you to the description of that class on the Acumen training website. The Acrobat class schedule is on the next page.

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

## PostScript Classes

| | |
|---|---|
| **PostScript Foundations** | October 7 – 11 |
| **Advanced PostScript** | November 11 – 15 |
| **PostScript for Support Engineers** | October 21 – 25 |
| **Jaws Development** | |

For more classes, go to www.acumentraining.com/schedule.html

**PostScript Course Fees**

PostScript classes cost $2,000 per student.
These classes may also be taught on your organization's site.
Go to www.acumentraining.com/onsite.html for more information.

Registration →
Acrobat Classes →

# Acrobat Class Schedule

These classes are taught quarterly in Costa Mesa, California, and on corporate sites. Clicking on a course name below will take you to the class description on the Acumen Training web site.

| | |
|---|---|
| **Acrobat Essentials** | October 3 (½-day, morning) |
| **Interactive Acrobat** | |
| **Creating Acrobat Forms** | October 3 (½-day, afternoon) |
| **Troubleshooting with Enfocus' PitStop** | |

**Acrobat Class Fees**

*Acrobat Essentials* and *Creating Acrobat Forms* (½-day each) cost $180.00 or $340.00 for both classes. *Troubleshooting With PitStop* (full day) is $340.00. In all cases, there is a 10% discount if three or more people from the same organization sign up for the same class.

Registration ->

Return to Main Menu

# Contacting John Deubert at Acumen Training

**For more information**

For class descriptions, on-site arrangements or any other information about Acumen's classes:

**Web site:** http://www.acumentraining.com **email:** john@acumentraining.com

**telephone:** 949-248-1241

**mail:** 25142 Danalaurel, Dana Point, CA 92629

**Registering for Classes**

To register for an Acumen Training class, contact John any of the following ways:

**Register On-line:** http://www.acumentraining.com/registration.html

**email:** registration@acumentraining.com

**telephone:** 949-248-1241

**mail:** 25142 Danalaurel, Dana Point, CA 92629

**Back issues**

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

Return to First Page

# What's New at Acumen Training?

## See you at Seybold!

I'll be teaching a pair of Acrobat classes at the Seybold Seminars (starts on September 9) in San Francisco again this year. Come on by and say "hi," if you're in town.

Monday, 9/9, morning session - *PDF for Prepress*
Tuesday, 9/10, afternoon session - *Creating Acrobat Forms*

Go to the Seybold Seminars website for more information on the show.

## *Creating Acrobat Forms*

Have you bought one for your poodle, yet?


Why not? *(Acrobat Forms* has been shown to promote healthy gum tissue in most breeds of show dogs.)

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 compell you to explain the disadvantages of being human to your household pets?

**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