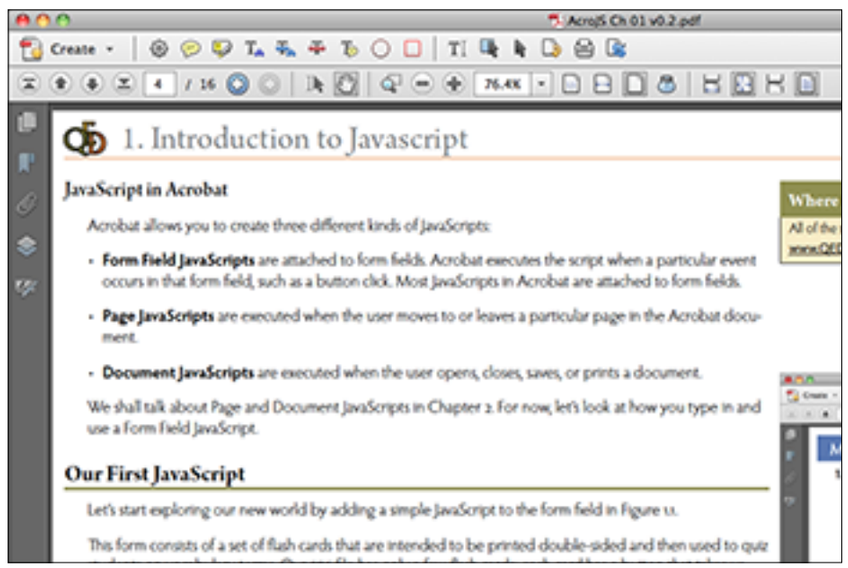
Table of Contents



Journal feedback: suggestions for articles, questions, etc.

Logging Debugging Information

Sample Files

As usual, the PostScript code for this month's article is available on the Acumen Training Resources page. Look for the file *LogFile.ps*.

Debugging PostScript code uses an awful lot of the debugging techniques that we otherwise left behind decades ago, when source code debuggers and similar tools became readily available. One common technique, when figuring out where a piece of code is going wrong, is to write diagnostic information to a log file or, if the interpreter I'm using doesn't let me create files, to stdout.

In this issue, I'm going to present a small set of PostScript procedures that I use to log diagnostic information. These procedures let me send text to either stdout or to a separate file on the rip's hard disk, if the PostScript device makes that possible.

The Simplest Case

At its most basic, you really don't need any procedures to send information to stdout. A common technique for locating a bug in PostScript code is to emit a series of marker strings, like this:

```
100 200 moveto
... Many lines of PostScript Code
(A) =
... Many lines of PostScript Code
(B) =
... Many lines of PostScript Code
(C) =
... Many lines of PostScript Code
... etc.
```

When we execute the code, the log file (or whatever is the target for stdout on that interpreter) will receive letters A, B, and so forth until the error takes place:

```
A
B
Error: typecheck  OffendingCommand: get
```

From the output, we can see that the error occurs somewhere between the `(B)` and `(C)` markers in the code.

So, Why Do We Need Procedures?

What I want to be able to do, that a simple series of "=" calls won't give me, are two things:

- I'd like to be able to redirect the diagnostic information to a my own log file on the RIP's hard disk, if the interpreter allows it.
- If I'm sending to stdout, I'd like my diagnostic information to be isolated from any other messages that the RIP emits, so I can see them easily. That is, rather than:

```
[Page 1]
A
B
[Page 2]
C
%== Warning: ThongUndies-Bold not found, substituting Courier
D
[Page 3]
... and so on
```

I'd prefer my diagnostic information to be at the end:

```
[Page 1]
[Page 2]
%== Warning: ThongUndies-Bold not found, substituting Courier
```

```
[Page 3]
=======Begin Log Output========
A
B
C
D
```

This isn't just tidiness; if I'm debugging a sizable PostScript stream, my diagnostic log entries can be scattered among a very large wad of irrelevant (to me) output. It's much more convenient if the log text is all together at the end of the stdout stream.

I think of the four procedures I present here as the *LogFile procset,* although I don't define them as a formal ProcSet resource; I never felt the need to do so, somehow.

The Procedures

The LogFile procset consists of four procedures that let you open, close, and write data to the log file. The procedures are

LogOpen	% (filename) => – – –	Opens the log file (or stdout)
LogClose	% – – – => – – –	Closes the log (duh!)
LogWriteString	% (str) => – – –	Writes a string to the log
LogWriteData	% anyObj (label) boolEOL => – – – Converts the object to a string and writes it to the log, preceded by the label and followed by an end-of-line of boolEOL is true.	

Here are the definitions:

LogOpen The *LogOpen* procedure, as you would expect, opens the log file. It takes the name of the log file, as a string, from the stack and opens that file with write permission. If the name is an empty string `()`, it creates a NullEncode filter attached to a 10 k string; we'll use this as a virtual file to which we can write our diagnostic information.

```
/LogOpen  % (filename) => ---
{  /lfLogFileName exch def                % Save the file name
   /lfUseStdOut lfLogFileName () eq def   % Test the name to see if it's ()
   lfUseStdOut                            % Is the name ()?
   {
      /lfOutputString 10000 string def    % Yes: create a 10k string...
      /lfLogFile lfOutputString /NullEncode filter def  % ...& attach NullEncode
   }
   {
      /lfLogFile lfLogFileName (w) file def  % No: Open the named file
   } ifelse
   % Write a "Beginning of Log" string, using the LogWriteString procedure
   (=== Begin Log File Output ===\n) LogWriteString
} bind def
```

LogClose *LogClose* closes the log file and then, if the log file was the NullEncode filter, writes the underlying string buffer to stdout. Note that this procedure uses a variable, *lfUseStrOut,* that was created by the *LogOpen* procedure.

```
/LogClose % --- => ---
{  (==== End of Log File Output ===\n) LogWriteString   % Write a final string
   lfLogFile closefile                    % Close the file
   lfUseStdOut { lfOutputString = } if % Write the output string to stdout,
} bind def                                %                  if appropriate
```

LogWriteString

The *LogWriteString* procedure writes a string to the log file. It's pretty simple:

```
/LogWriteString  % (str) => ---
{ lfLogFile exch writestring } bind def
```

LogWriteData

And, finally, *LogWriteData* takes an object of any sort, a label string, and a boolean and writes the object to the log as a string, preceded by the label. If the boolean is true, the procedure follows the data with a newline.

Thus,

```
96 (The height is ) true LogWriteData
```

would result in the following text being written to the log:

```
The height is 96<nl>
```

Here's the definition:

```
/LogWriteData     % obj (Label) boolAddEOL => ---
{  3 1 roll                % Roll the boolean to the bottom of the stack
   LogWriteString          % Write the Label to the log
   30 string cvs           % Convert the data to a string...
   LogWriteString          % ...and write it to the log
   { (\n) LogWriteString } if   % Write a newline if the boolean was true
} bind def
```

You may feel queasy about the `30 string` in the above code; It looks as though this would be an ongoing memory leak, because every time we execute *LogWriteData* we allocate a new string. This turns out to be okay; we create the string and immediate hand it to the *cvs* operator, after which it becomes inaccessible. In cases like this, PostScript immediately reclaims the VM used by the string.

To use the log procset to examine a piece of problematic PostScript, we do the following:

1. Put a call to *LogOpen* at the beginning of the code.
2. Add a call to *LogClose* at the end of the code.
3. Add calls to *LogWriteData* or *LogWriteString* as needed to examine the execution of the PostScript code.

The code would end up looking like this:

```
(/MyTestLog.log) LogOpen
% Here's your test code
... PostScript code
(A) LogWriteString
... PostScript Code
(This is a test) LogWriteString
243 (Here's a number: ) true LogWriteData
LogClose
```

The log file would have the following in it, following whatever text is generated by the PostScript program itself:

```
=== Begin Log File Output ===
A
This is a test
Here's a number: 243
=== End of Log File Output ===
```

Final Notes

Distiller Acrobat Distiller 9 and X will not let your PostScript code create files on the local hard disk. Thus, if I absolutely must have a separate log file, I use Distiller 8 to test my code. If I'm using a later version of Distiller, then I send my diagnostic information to stdout.

So, When Do I Use It? Truth be told, I don't use this procset too often; mostly, I just scatter =’s through my code as needed. I press the procset into service when I'm working on something that needs extensive debugging. I used it a lot, for example, when I was working on the examples for the Variable Data PostScript class.

Submitting Form Data

I'm a big fan of Acrobat.com. In particular, when I'm sending a form to a collection of people, gathering information about their training-related opinions, desires, and fantasies, I pretty much always use Acrobat.com to distribute the form and collect the responses.

This Article Assumes…

…that you have some knowledge of creating Acrobat forms. In particular, I assume you can create a basic form with text fields, drop-down lists, and buttons.

However, there are some forms that I just post on Acumen Training website without a specific distribution list in mind. In this case, I don't particularly want Acrobat.com to maintain information about the form. So, how do you collect the data from such a form?

Well, you must create a Submit button; this is a button (actually named anything you like) that, when clicked, sends the form data to a url or email address.

If you send the data to a url, you need to have a script at that site waiting for the form data; that script should read the data and do something useful with it, usually place it in a database. That's often more trouble than I want to bother with; it's usually better to just have the form email the user's responses to me. I receive the message with the form data attached and I can do whatever I want with it. (Often, I just eyeball the data and make a couple annotations in an electronic notebook I keep.)

Creating a button that will send form data to you as an email attachment is really easy, so today I'm going to show you how to do it.

Strap in and let's get started.

Creating a Submit Button

Again, our goal is to create a button that, when clicked will email a form's data to us in one of two ways:

- As an *FDF* ("form data format") file.

  This is a small file that contains the user's responses to all of the form fields in the document. We can then

import this file into our copy of the same form to see user's data; we'll see how to do this later in the article.

- A copy of the entire PDF file, with all of the filled-in form fields in place.

  We can open this file in our own copy of Acrobat and inspect the responses.

We tell a form button to do this by attaching a pre-defined Submit action to its Mouse-Up event.

Our Example File

In this article, we'll work with the document in Figure 1, turning its "Sign me up" button into a Submit button that will email us the user's responses to the other form fields (two text fields and a drop-down menu).

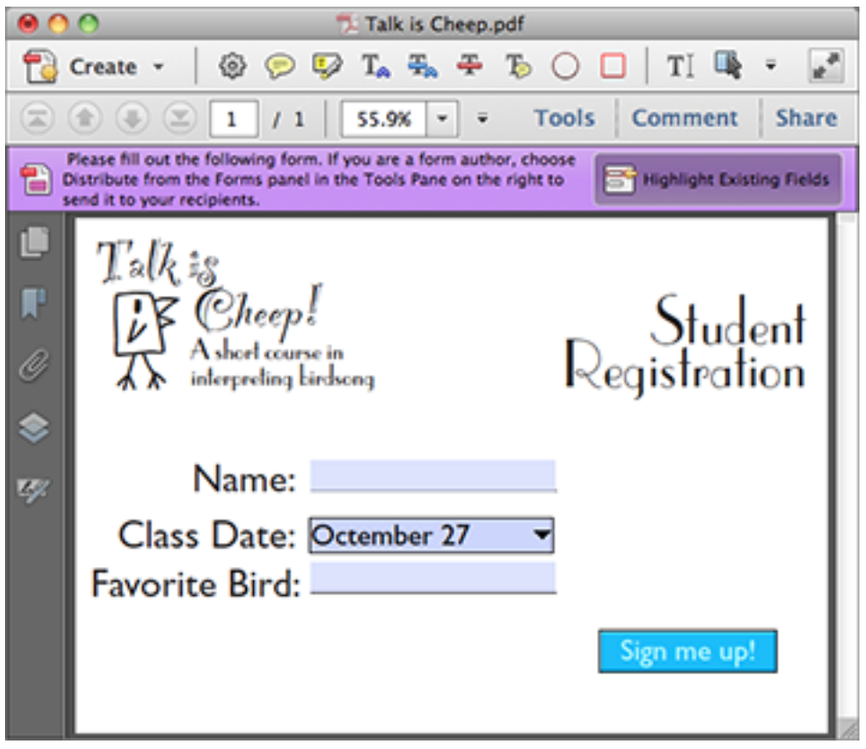


Figure 1. We are going to turn this form's *Sign me up!* button into a functioning Submit button.

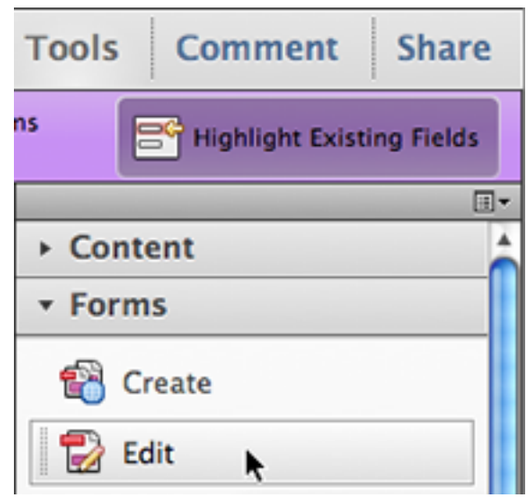


Figure 2. Select the Edit tool in the Forms panel.

Creating the Submit Action

Here's how we attach the Submit Form action to the button's Mouse Up event:

1. In the Tools pane's Forms panel, click the Edit tool (Figure 2).

   Acrobat will drop into the Form editor (Figure 3, next page). This looks just like the normally-displayed PDF page, except:

   - Each of the page's form fields is represented by a rectangle containing the field's name.
   - The toolbars have changed, the normal Quick Tools toolbar being replaced with a set of forms-

related tools.

2. Double-click on the *Sign me up* Button field.

   Acrobat will display the Button Properties dialog box (Figure 4).

3. Go to the Actions tab in the Button Properties dialog box and choose the following settings in the pop-up menus (these are shown in Figure 4):

   - In the Select Trigger pop-up menu, select Mouse Up.
   - In the Select Action menu, select Submit a Form.

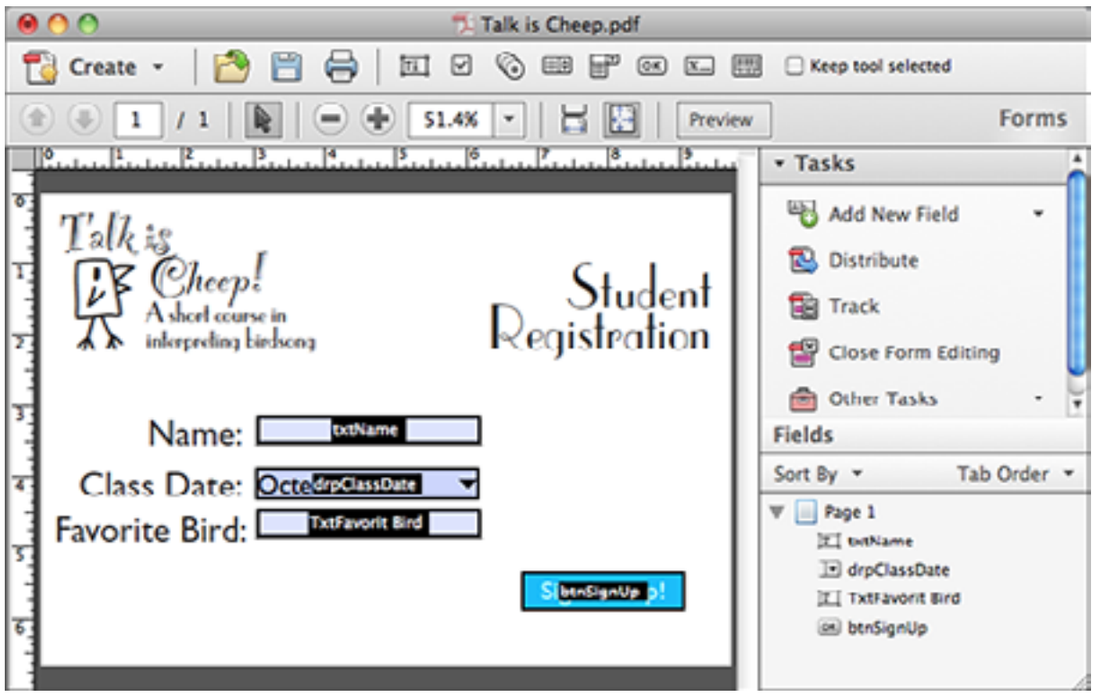


Figure 3. In the Form Editor, each of a page's form fields is displayed as a rectangle containing the field's name.

5. Click the Add button.

   Acrobat will display the awkwardly-named Submit Form Selections dialog box (Figure 5, next page).

6. Type your email address, preceded by the suffix "mailto:" into the URL text field, something like

   ```
   mailto:john@acumentraining.com
   ```

7. Select either the FDF or PDF radio buttons, according to how you want your data sent to you.

   - FDF files contain only the user's response to the form fields and are very much smaller

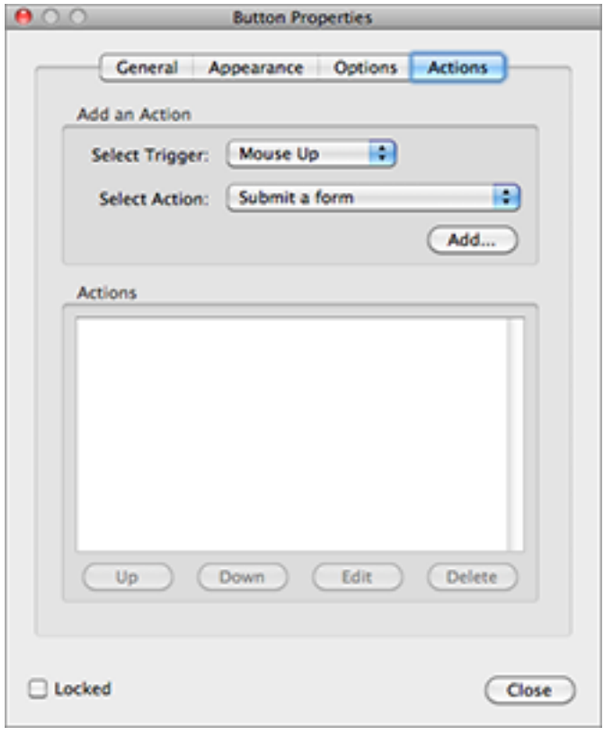


Figure 4. The Actions tab in the Button Properties dialog box lets us specify what the Button should do when clicked.

than the original PDF document, usually a kilobyte or less in size. However, they require a (very little) bit of work to examine their contents. We'll see what that entails later in the article.

- The PDF file selection means just that, you'll get the user's copy of the PDF form, complete with filled-in form fields.

8. Click the OK button, returning you to the Button Properties dialog box and then the Close button, returning you to the Form Editor.

9. To leave the Form Editor and return to normal viewing of your pdf document, click the Close Form Editing button in the Forms pane (Figure 6).

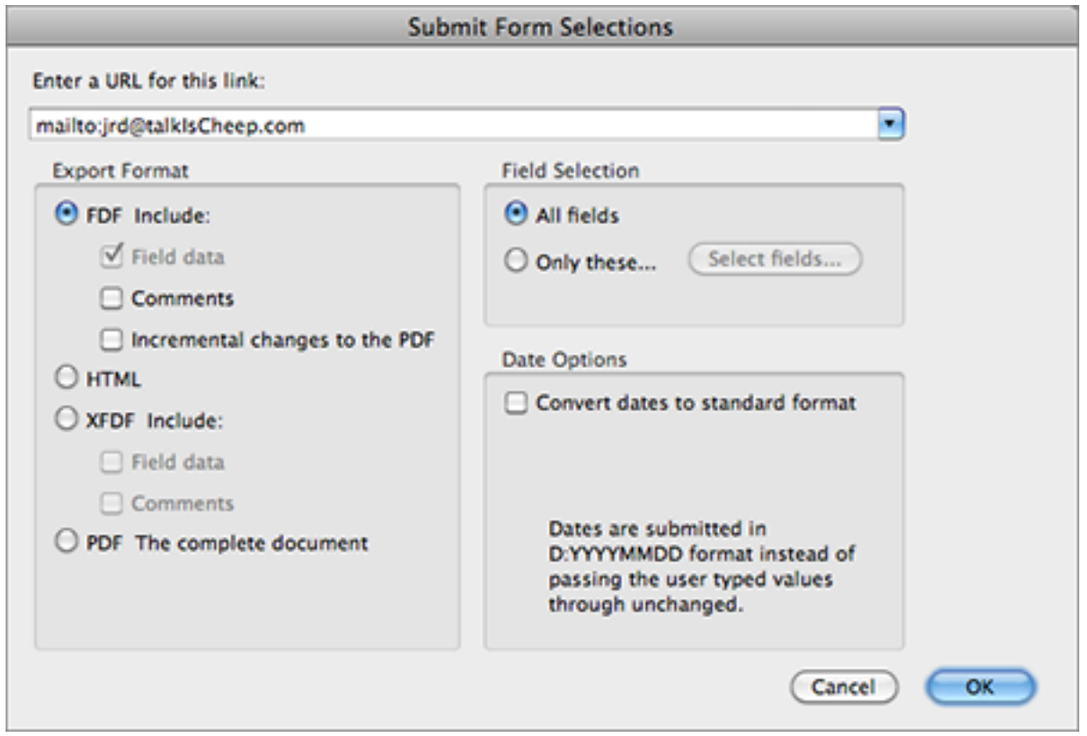


Figure 5. In the *Submit Form Selections* dialog box, pick the FDF or the PDF radio buttons.

That's it. Now you can post this PDF form to your web page or do anything else you wish. When somebody fills it out and clicks the Submit button, you will receive an email with an attached PDF or FDF file.

So, What Do I Do with FDFs?

If you chose to have your file sent back to you as a PDF file, there's no special handling you need to do when you receive a user's response; just open the file up in your copy of Acrobat and look at what they typed into the fields.

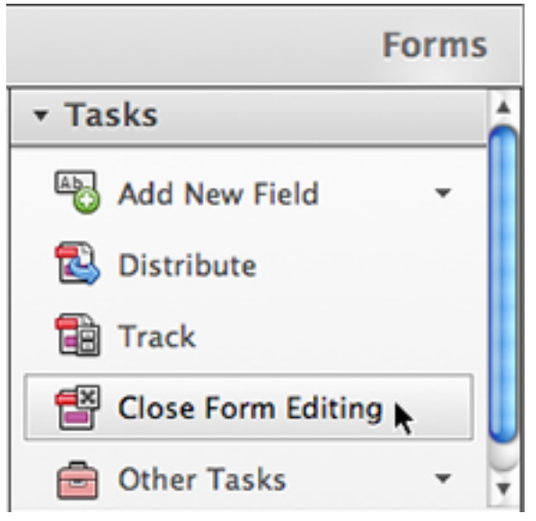


Figure 6. Return to the normal view of your PDF file by selecting Close Form Editing in the Forms Tasks pane.

On the other hand, an FDF file is not intended to be a standalone document. To read the responses encoded in an FDF file, you need to import it into your copy of the form. Happily, this is really easy; in fact, it's a one-step process:

1. In the Forms pane, select *More Form Options* > *Manage Form Data* > *Import Data* (Figure 7). Then, select the FDF file in the resulting Picka-a-File dialog box.

That's it. Acrobat will load up your copy of the form with the values encoded in the FDF file.

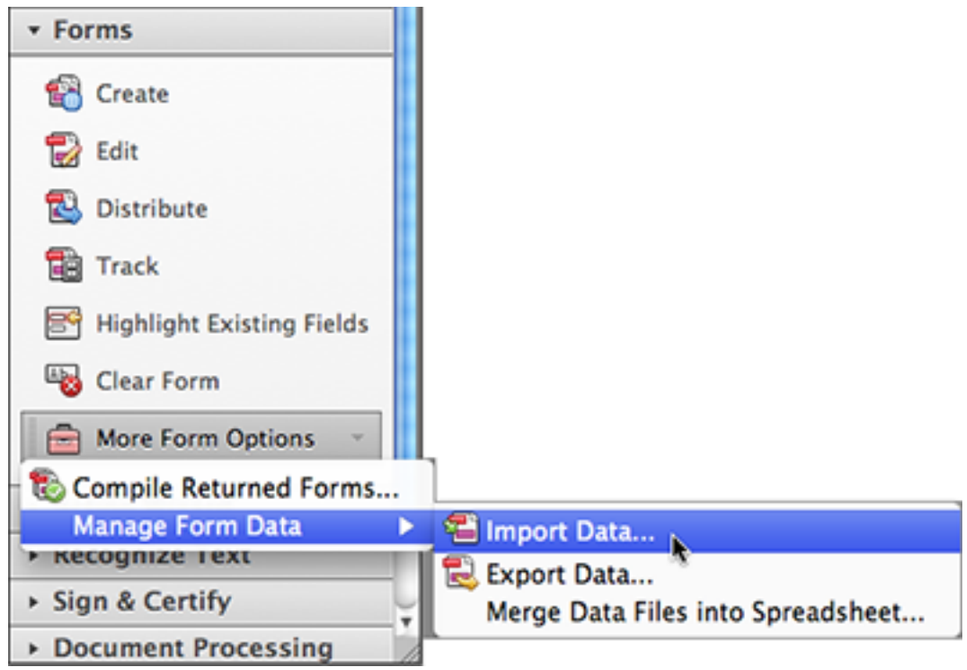


Figure 7. To view the data encoded in an FDF file, open your copy of the PDF form and then import the FDF file. Acrobat will populate the PDF form fields with values it finds in the FDF file.

Submit Button vs. Acrobat.com

Acrobat.com is at its best with forms that you're sending to a specific group. For example, I often send PDF evaluation forms to students who have attended the beta version of a new class. It is most useful to distribute this and collect the responses using Acrobat.com

On the other hand, forms that you are distributing to an indeterminate group—perhaps a form that you send out only when someone asks for it or, in my case, a form residing on a page in an Acumen Journal—then it's best to provide that form with a Submit button. Most of the PDF forms I distribute are in this category. For example, I have a form that I send to people that have asked about a customized PostScript or PDF class; they can check the topics in which they are interested then click the form's Submit button to send me the list.

PDF Nuggets

PDF Character Encoding

In designing PDF, Adobe applied many lessons learned from their PostScript experience, often greatly simplifying the way you do common tasks. For example, every time you use a font in either PostScript or PDF, you need to re-encode the font so its character encoding matches that of the strings within the file. In PostScript, this entailed a moderately complicated piece of code, like so:

```
/MacEncoding
[ /.notdef /.notdef /.notdef /.notdef /.notdef /.notdef /.notdef
... many, many character names here ...
/tilde /macron /breve /dotaccent /ring /cedilla ] def
```

```
/Helvetica findfont dup length dict begin
currentdict copy pop
/Encoding MacEncoding def
/Helvetica-Mac currentdict end definefont pop
```

PDF lets you specify common character encodings by name:

```
<<
   /Type      /Font
   /Subtype       /Type1
   /BaseFont  /Times-Roman
   /Encoding  /MacRomanEncoding     % Could also have been "/WinAnsiEncoding"
>>
```

This makes life very much easier. The PostScript code is generalized, letting you set the character encoding to anything you wish. PDF is optimized for the common case.

Schedule of Classes, July 2011– October 2011

At right are the dates of Acumen Training's upcoming classes. Clicking on a class name will take you to the description of that class on the Acumen Training website.

O.C. and On-Site

These classes are taught in Orange County, California and on-site at corporate sites world-wide.

Please see the Acumen Training web site for more information, including an up-to-date schedule.

Class Fee

Classes cost $2,000 per student, with the following exceptions:

- *Troubleshooting PostScript* $1,500
- *Support Engineers' PDF* $1,000

There is a 10% discount for signing up three or more students.

Note that 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.

PDF Classes

PDF 1: File Content and Structure		Aug 22–25	Oct 3–7
PDF 2: Advanced File Content			
Support Engineers' PDF	Jul 21–22		Sept 15–16

PostScript Classes

PostScript Foundations		Aug 8–12	Sept 26–30
Advanced PostScript		Aug 15–18	
Variable Data PostScript	Jul 25–29		
Troubleshooting PostScript	Jul 18–20		Sep 12–14

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

What's New at Acumen Training?

An Update to the JavaScript Book is Coming

I'm working on an update to the old *Extending Acrobat Forms with JavaScript* book. I'll be self-publishing this as an eBook and it will be available later this year. I'll post an exact date in the next *Journal* issue.

In the meantime, if you want to be notified when the book's ready, drop me an email at john@acumentraining.com; put "Javascript book" somewhere in the subject.