Table of Contents

Acumen Training

The JavaScripter: Creating a Simple Blinking Button

This month, we start *The JavaScripter,* a series of occasional articles that explore some of the interesting and useful things you can do with JavaScript in Adobe Acrobat. These are intended for the advanced beginner who has at least some JavaScript experience, about as much as you would get from my new book, *Extending Acrobat Forms With JavaScript.* Nonetheless, there will always be enough background information so that the article will be useful even if you are a complete beginner.

For our first article in the series, we shall see how to make a simple blinking button, like the one at right. This script requires Acrobat 5 or later; if you are reading this *Journal* issue with Acrobat 4, the button won't blink.

Let's look at how to do it.

Next Page ->

Overview

The project's starting point is a simple, non-blinking button initially colored a dark red, like the top illustration at right. To make this button blink, we will change its color periodically—every half-second or so—toggling between the button's initial color and a bright red.

The *setInterval* method

We are going to do this with a call to the Acrobat JavaScript *app* object's `setInterval` method:

```
timeoutObj = app.setInterval("JS Code", millisecs)
```

The `setInterval` method tells Acrobat to execute the JavaScript code that's in the quotes once every so many milliseconds. Thus, the following line:

```
timeoutObj = app.setInterval("app.beep()", 1000)
```

will cause Acrobat to beep once per second.

The `setInterval` method returns something called a *timeout* object. This is the object which you can later use to turn *off* the periodic process you start with `setInterval`. You do this by calling the *app* object's `clearInterval` method:

```
app.clearInterval(timeoutObj)
```

This halts the specified `setInterval` process. (That periodic beep gets *mighty* annoying after an hour or two.)

Again, this article's discussion presumes you have had at least some JavaScript experience. In particular, I'm assuming you know some of the terminology: "method," "object," etc.

If you don't know JavaScript, you can still follow the directions and type in the scripts; everything will work as described.

Next Page ->

The Document JavaScript

We want our button to start blinking as soon as we open the document, so we are going to make our call to `setInterval` in a *Document JavaScript;* you may remember that document JavaScripts are executed by Acrobat when the document opens.

What You Need to Start

Before you start, the PDF file with the button must be open in Adobe Acrobat. Also, you must know the JavaScript name of the button that should blink; in our sample code, we shall assume the button's name is "btnClickMe." (See your favorite Acrobat forms book for a reminder of how to create and name buttons.)

Entering the JavaScript

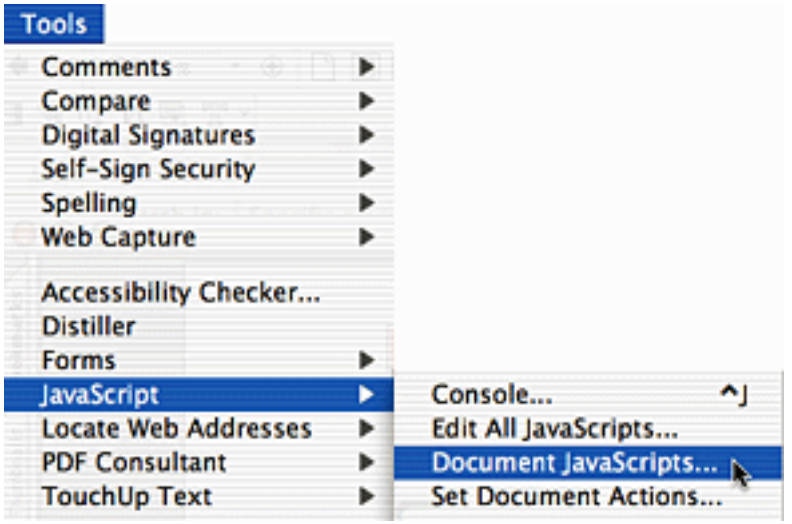
To type in the Document JavaScript, start with the document open in Acrobat and do the following:

If you want to follow along, the file *BlinkingBtn.pdf* on the Acumen Training Resources page has a button you can turn into a blinker.

1. Select *Tools>JavaScript> Document JavaScripts.*

   Acrobat will present you with the *JavaScript Functions* dialog box (next page).

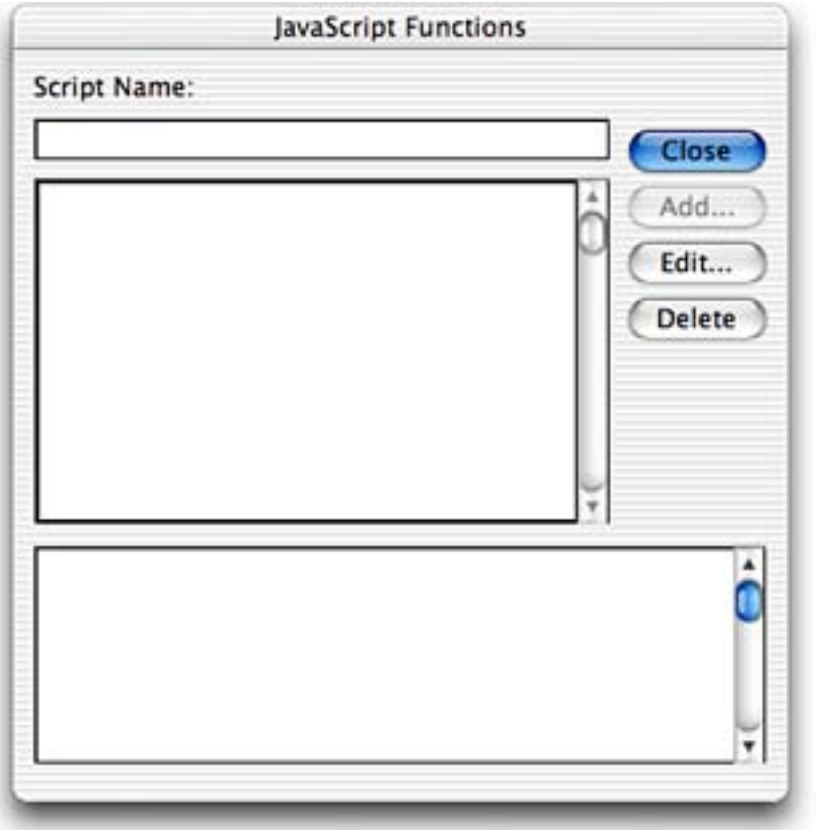
Next Page ->



2. Type a script name into the *Script Name* field (I suggest "ToggleButton") and click the *Add* button. What you choose as a script name is not too important; anything short and descriptive will do.

   Acrobat will present you with the *JavaScript Edit* dialog box, below. This will have a bit of text already entered into it, as in the illustration.

Next Page ->



Remember that the double slashes ("//") denote a comment that is ignored by JavaScript. You don't need to type in anything from the // to the end of the line.

3. Erase the initial text in the *JavaScript Edit* dialog box and type in the following script. (We shall step through the details of what this is doing shortly.)

```
var btnOn = false          // Create a Boolean variable, set it to false

function ToggleButton()    // Start of a function definition
{
  var btn = this.getField("btnClickMe")  // Get the button "btnClickMe"

  btnOn = !btnOn           // Reverse the value of btnOn
  if (btnOn)               // Is btnOn now true?
    btn.fillColor = color.red  // Yes: set the button's color to red
  else                     // Otherwise...
    btn.fillColor = ["RGB", .5, 0, 0]    // Set the color to dark red
}                          // End of our function

// Now turn on the blinking:
if (app.viewerVersion >= 5) // But only if our Acrobat is version 5 or later
  var blinkObject = app.setInterval("ToggleButton() ", 500)
```

When you are finished, click the *OK* button, returning to the *JavaScript Functions* dialog box.

4. Click the *Close* button in the *JavaScript Functions* dialog box, returning to the Acrobat document.

5. Save and re-open the document and your button will start blinking.

Next Page ->

Stepping Through the Code

This JavaScript does three things:

- It defines a Boolean (*i.e.,* true-or-false) variable named "btnOn."
- It defines a function named "ToggleButton." This function reverses the value (true becomes false, and *vice versa)* of the *btnOn* variable and then sets our blinking button's color to either bright or dark red, depending on *btnOn's* new value.
- It executes the *app.setInterval* method, starting our button to blink.

Line by Line

```
var btnOn = false
```

Here we create the variable *btnOn.* This variable has a Boolean value that we shall use to indicate whether the button is currently on (bright red) or off (dark red). Each time we change the color of the button, we shall reverse the value of this variable.

```
function ToggleButton()
{
```

We now start our definition of `ToggleButton`.

```
var btn = this.getField("btnClickMe")
```

`ToggleButton` first gets a reference to our button (named "btnClickMe" in this example) and assigns the reference to the variable, *btn.* Remember that "this" refers in this context to the current document. (Think of it as meaning "this document.")

Next Page ->

```
btnOn = !btnOn
```

This cryptic looking line reverses the value of our Boolean variable, *btnOn.* In JavaScript, the exclamation point (pronounced "not") reverses the sense of whatever Boolean value follows it. Thus, "!btnOn" (to be read, "*not* btnOn") has the value opposite that of *btnOn:* true if *btnOn* is false, false if the variable is true.

```
if (btnOn)
  btn.fillColor = color.red
else
  btn.fillColor = ["RGB", .5, 0, 0]
```

If you have read the JavaScript book, you have seen code similar to this before. Loosely interpreted, this block of code says: "If *btnOn* has a value of true, then set the button's fill color (that is, its background color) to red; otherwise, set the button's fill color to an RGB value of .5, 0, 0."

Color component values in Acrobat JavaScript vary from *0* to *1,* with zero meaning "off."

```
}
```

The close brace ends our function definition.

Whenever we want to reverse the color of our button, we need simply call `ToggleButton()`.

Next Page ->

```
if (app.viewerVersion >= 5)
  var blinkObject = app.setInterval("ToggleButton()", 500)
```

Finally, we start our button blinking with a call to the *app* object's `setInterval` method. We only want to do this if the document is being viewed with a recent version of Acrobat; `setInterval` didn't exist in Acrobat 4 or earlier.

The above lines of JavaScript first check to see if our current viewer (Acrobat, Reader, etc.) has a version number greater than or equal to 5. If so, our script tells Acrobat to execute the JavaScript statement "`ToggleButton()`" every 500 milliseconds. Every half-second, Acrobat will call our `ToggleButton` function, reversing the color of the *Click Me* button.

Our button is now blinking!

Timeout Objects

Note that `setInterval` returns something called a *Timeout object;* this is an object that represents our recurring process within this and other JavaScripts in our Acrobat document. In our case, we store this object in a variable named "blinkObject." We will use this object in the next section to halt the blinking.

Customizing the Script

To customize the above script to your own form, you'll need to make the following changes as appropriate to your form:

- In the definition of `ToggleButton`, change the button name in the call to `getField` to the name of your button.
- Also in `ToggleButton`, change the "on" and "off" colors to whatever you want; see the note below on specifying color in JavaScript.

Next Page ->

- In the call to `setInterval`, change the interval to the number of milliseconds you wish.

Specifying Color

Every form field has three color properties that you can usefully change in a JavaScript:

fillColor The background color of the field.

borderColor The border color of the field.

textColor The color used for the field's text.

You set these colors by getting a reference to the form field (using `app.getField`) and then assigning colors to these properties:

```
var myFld = app.getField("myFieldName")
myFld.fillColor = color.red
myFld.borderColor = color.blue
myFld.textColor = ["RGB", .5, 1, .5]
```

Acrobat JavaScript defines names for commonly-used colors:

JavaScript Color Names

color.transparent	color.red	color.cyan
color.black	color.green	color.magenta
color.white	color.blue	color.yellow
color.gray	color.ltGray	color.dkGray

Next Page ->

Acumen Journal: Acrobat User 10

If you want a color other than one of those listed above, then you may use an array that contains an RGB, CMYK, or grayscale color, as in the *textColor* line, above. The first element of the array is a string that indicates what kind of color you want: "RGB", "CMYK", or "Gray". The remaining elements are the actual color specification, each a value from *0* to *1.* (A value of *0* means "no light" or "no ink.")

Turning off the Blinking

At this point, we've taught Acrobat how to make our button blink; it would seem as though we are done. However, if we leave things as they are, something odd happens: when we close the document, we get a series of JavaScript errors, reported in the JavaScript console as at right.

My theory as to why this happens is that, when you close the document, Acrobat cleans up the memory occupied by the various JavaScript objects and functions (such as `ToggleButton`) *before* it cancels any recurring processes specified by `setInterval`. Thus, every half-second, Acrobat is still trying to execute the `ToggleButton` procedure, which has already been destroyed by the closing of the document.

To avoid this, we need to explicitly turn off our `setInterval` process when the document closes. We do this with a JavaScript that calls the *app* object's `clearInterval` method:

```
app.clearInterval(myTimeoutObject)
```

Next Page ->

The `clearInterval` method takes a timeout object, created earlier by a call to `setInterval`, and turns off the corresponding recurring process. We shall call `clearInterval` in a *Document Action* associated with the *Document Will Close* event.

Document Action Script

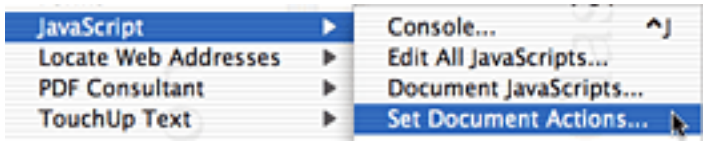
Document Actions are associated with events that affect the document as a whole: opening, closing, etc. We are going to attach a JavaScript that turns off our blinking button with the *Document Will Close* event, that occurs just before the document closes.

Attaching the Script

To attach the JavaScript to the document's *Will Open* event, start with the Acrobat file open, and do the following:

1. Select *Tools* > *JavaScript* > *Set Document Actions…*

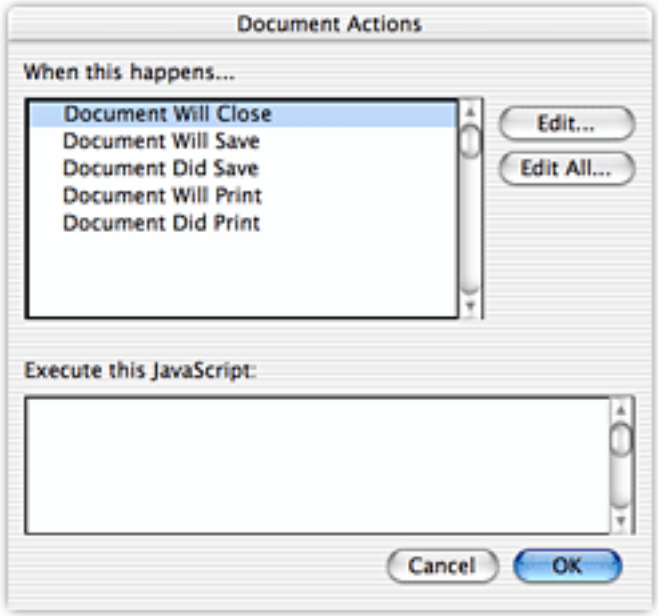
   Acrobat will present you with the *Document Actions* dialog box (next page).



Next Page ->

2. In the *When this happens* list, select *Document Will Close* and then click the *Edit* button.

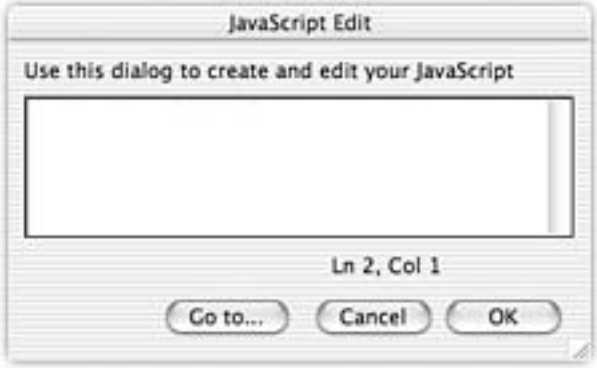
   Acrobat will present you with the same *JavaScript Edit* dialog box we saw when we typed in our Document JavaScript.



3. Type the following two lines of JavaScript into the dialog box:

```
if (app.viewerVersion >= 5)
  app.clearInterval(blinkObject)
```

   You may remember that *blinkObject* was the variable into which we stored the *Timeout* object returned by our earlier call to `setInterval`. Again, we want to call `clearInterval` only if we have a recent version of Acrobat, so we check for that with an `if` clause.



4. Exit out of all dialog boxes until you are back to the Acrobat document.

That's it; we're done. Nothing dramatic happens as a result of typing in this Document Action JavaScript; we simply won't get that annoying set of errors when we close the document.

Next Page ->

Conclusion

As you can see from this example, you can add some very nifty effects to your Acrobat forms with very simple JavaScripts. The *JavaScripter* articles will try to concentrate on this: useful tasks that can be carried out with simple JavaScripts. I will be going out of my way to not duplicate examples from my JavaScript book, so these articles will serve as additional "lessons" for readers of that book.

If there is something you would like to appear in the *JavaScripter* articles, feel free to drop me an email.

Return to Main Menu

A PostScript Profiler (Sort of)

A couple months ago, a student asked me about writing a PostScript profiler, something that would tell you where a PostScript program is spending its time so you can improve its execution speed. I know of no such beast offhand, but it sparked a discussion about how one might go about at least counting the number of times a PostScript program executes each of the PostScript operators. Not the same as a real profiler, but it might be of some diagnostic use, we thought.

This month, we'll look at a way of doing this.

We are going to define a PostScript header that can be placed at the beginning of any PostScript code; the header redefines all of the PostScript operators in *systemdict* so that each operator increments a counter when executed. The header also defines a procedure that will report the results if you execute it after the target PostScript.

As an example, at right is part of the report that resulted when I tested some Adobe Illustrator output. The report is only minimally formatted and unsorted. We may look at PostScript sorting routines in a future *Journal* article, if anyone's interested. (I have a swell Quicksort implementation.)

This topic will give us a chance to look at some of the ins and outs of redefining PostScript operators.

```
setglobal: 4
if: 4374
array: 4
dict: 63
currentglobal: 3
dup: 490
forall: 8
readonly: 1
put: 62
for: 1
exec: 14
get: 45
readline: 2053
dictstack: 1
...
```

Next Page ->

Acumen Journal: PostScript Tech

Redefining Operators

In principle, redefining PostScript operators is relatively simple: just define a procedure with the same name as the operator. Since that procedure must invariably go into *userdict* or some other dictionary higher on the Dictionary stack than *systemdict,* our redefinition will be seen by the PostScript interpreter at name lookup time.

Thus, the following PostScript code:

```
/moveto { 1.1 mul moveto } bind def
100 100 moveto
```

will leave the current point at the position *100,110.* Note that our redefinition of *moveto* itself calls *moveto.* This is not recursive, as it at first seems, because we handed our procedure body to *bind* before we executed *def.* The *bind* operator replaced the name "moveto" in the procedure body with the definition of *moveto* at the time that the *bind* took place, that is, before our redefinition had occurred with *def.*

If you paste the redefinition of *moveto* at the beginning of someone else's Postscript code, that output would have all calls to *moveto* displaced by 10% along the *y* axis. (Why would you want to do that? I don't know; it wouldn't even be very amusing as a prank, as far as I can see.)

Next Page ->

Making Your Own *systemdict*

The problem with the simple *moveto* redefinition above is that if you were to paste it in front of randomly-chosen PostScript output from the likes of, say, *QuarkXpress,* it may well fail to be called. Many applications and drivers bypass any such redefinitions by going directly to *systemdict* for their Postscript definitions:

```
/m systemdict /moveto get def
```

Unfortunately, this very effectively ties the name *m* to the definition that *moveto* has in *systemdict,* not in whatever dictionary contains our redefinition.

The workaround is to create your own dictionary and call *that* "systemdict." You can copy everything from the original *systemdict* into your replacement and then add your redefinitions. Then place your *systemdict* and the original *globaldict* and *userdict* onto the dictionary stack; whenever anyone gets a definition out of *systemdict,* they'll get it out of *your* version of *systemdict.*

```
/systemdict0 systemdict def                % Create a new dictionary
/systemdict systemdict dup length dict copy def % Copy systemdict into it
systemdict begin                           % This is our systemdict
/moveto { 1.1 mul moveto } bind def        % Redefine moveto
globaldict begin
userdict begin
%===================
% QuarkXpress' PostScript goes here
% ...
/m systemdict /moveto get def              % This gets our redefinition
```

Next Page ->

Acumen Journal: PostScript Tech 17

Creating the Counter Procedures

Presume for the moment that we have defined a procedure called *IncrementCount* that takes an operator name from the operand stack and increments the execution count associated with that operator. (We'll look at *IncrementCount's* definition later.) What I want to do in my "profiler" is redefine each operator to a procedure that executes *IncrementCount* with that operator's name and then executes the original operator.

Thus, our redefinitions should look something like this:

```
/systemdict0 systemdict def
/systemdict systemdict length dict def
systemdict begin

/IncrementCount     % /name => ---
{
     ... some PostScript stuff
} bind def

/moveto { /moveto IncrementCount moveto } bind def
/add { /add IncrementCount add } bind def
... etc.
```

Making these procedure definitions will get tedious after about the 150th PostScript operator, so we'll want to automate the process.

Let's look at the full full PostScript code:

Next Page ->

Acumen Journal: PostScript Tech 18

The Code

As usual, this PostScript code is available on the Acumen Training Resources page. Look for the file *CountOps.ps.*

```
/str 64 string def                      % Scratch string
/dictstack0 countdictstack def          % Used for cleanup

/profileDict   <<                       % This dictionary holds our op counts
     systemdict { pop 0 } forall        % Each operator name is associated with
>> def                                  % the number of executions.


/IncrementCount          % /name   =>   ---
{    profileDict exch                   % => <<profileDict>> /name
     2 copy get                         % => <<profileDict>> /name count
     1 add                              % => <<profileDict>> /name count+1
     put                                % => ---
} bind def

/systemdict0 systemdict def                 % Save the old systemdict...
/systemdict systemdict length dict def      % and make a new one

% Load the new systemdict with replacements for the operators
systemdict0          % For each key-value pair in original systemdict
{    dup type /operatortype eq          % => /key  val  bool
     {    [ 2 index                     % => /key  val  mark  /key
            /IncrementCount cvx         % /key  val  mark  /key  <IncrementCount>
            4 -1 roll                   % /key  mark  /key  <IncrementCount>  val
          ]                             % => /key  [/key <IncrementCount> val]
          cvx                           % => /key  {/key <IncrementCount> val}
     } if
     systemdict 3 1 roll put
} bind forall
```

Next Page ->

Acumen Journal: PostScript Tech 19

Some Utilities

```
/StartProfiler      % Call this immediately before the PS code you are counting
{    systemdict begin         % Again, this is our systemdict
     globaldict begin
     userdict begin
} bind def


/StopProfiler       % Call this immediately after the PS code you are counting
{ countdictstack dictstack0 sub { end } repeat } bind def


% This proc prints a report to stdout; it just dumps the contents of profileDict
/ReportCalls
{    profileDict
     {    dup 0 eq
          { pop pop }
          { exch str cvs print (: ) print = }
          ifelse
     } forall
} bind def
```

Now, Let's try it out…

```
% ======== Begin Test ===========
StartProfiler
72 600 moveto 100 100 rlineto 0 -100 rlineto closepath
stroke showpage
StopProfiler
% ======== End Test Subject ===========

ReportCalls
```

Next Page ->

This program produces the following report, sent to *stdout:*

moveto: 1
showpage: 1
stroke: 1
rlineto: 2
closepath: 1

Step by Step

```
/str 64 string def
/dictstack0 countdictstack def

/profileDict   <<
     systemdict { pop 0 } forall
>> def
```

The program starts by defining three constants:

- *str* is a string buffer that we'll use in reporting our operator count values.
- *dictstack0* is the number of items on the dictionary stack at the start of the program; this should be *3* (coiunting *system-, global-,* and *userdict).*
- *profileDict* is a dictionary that holds our operator counts. In each key-value pair, the key is an operator name and the value is the number of times that operator has been called, initialized to zero.

  Note that we are populating this dictionary with a *forall* loop that piles alternating operator names and zeros on the operand stack. Open-double-angle-brackets (*i.e.,* "<<") puts a mark on the stack; our loop dumps the alternating names and zeros onto the stack; finally >> constructs the new dictionary, loading it with key-value pairs taken from the items on the stack.

Define IncrementCount

```
/IncrementCount          % /name   =>   ---
```

We define a procedure named *IncrementCount.* This procedure takes an operator name from the operand stack and increments the corresponding value in *profileDict.*

Next Page ->

Acumen Journal: PostScript Tech 21

```
{    profileDict exch                    % => <<profileDict>> /name
```

IncrementCount starts by getting *profileDict* and exchanging the dictionary and name on the operand stack; this puts them in the correct order for a later call to *get.*

```
2 copy get                               % => <<profileDict>> /name count
```

This incarnation of the *copy* operator takes an integer from the stack and copies the top *n* items on the stack to the top of the stack. In our case, we push an additional instance of *profileDict* and our operator name onto the stack. We then do a *get,* which gets the current count value associated with that operator name out of *profileDict.*

```
     1 add                               % => <<profileDict>> /name count+1
     put                                 % => ---
} bind def
```

IncrementCount ends by adding one to the operator count and then putting the result back into *profileDict.*

Create our own systemdict

```
/systemdict0 systemdict def              % Save the old systemdict...
/systemdict systemdict length dict def   % and make a new one
```

We save the original *systemdict* in a key-value pair named *systemdict0.* We then create a new dictionary, the same size as the original *systemdict,* and save this new dictionary with the name *systemdict.* Any future references to the name “systemdict” will yield our new dictionary, not the orginal *systemdict.*

Next Page ->

Acumen Journal: PostScript Tech 22

Load our systemdict

```
systemdict0
{
      ...
} bind forall
```

We are going to use a *forall* loop that steps through the original *systemdict,* doing the following for each key-value pair:

- Look to see if the key-value pair is an operator definition (by seeing if the value is of type *operatortype.)*
- If so, we shall construct a procedure body of the form

  ```
  { /key IncrementCount value }
  ```

  where *key* is the operator name and *value* is the operator definition.
- Put this procedure into our new *systemdict,* associated with the original operator name.

In stepping through the loop, remember that when *forall* is given a dictionary and a procedure, it puts each key-value pair onto the stack in turn and then calls the procedure.

Here's what happens within our loop:

```
{    dup type /operatortype eq      % => /key  val  bool
```

We duplicate the value of the current key-value pair and test to see if it's type is "operatortype." The *eq* leaves a Boolean value on top of the stack.

Next Page ->

Acumen Journal: PostScript Tech 23

```
{
```

This open brace starts an *if* clause that will be executed if the Boolean returned by *eq* is true, meaning the value is an operator definition.

```
[ 2 index                    % => /key  val  mark  /key
```

If the value is of type *operatortype,* then we put a mark on the stack (with the open bracket) and copy the key to the top of the stack with *2 index.*

Our plan is to construct an array containing the contents we want for our procedure and then convert that array to a procedure with *cvx.* As I said on the previous page, the procedure we are constructing from the key-value pair should be of the form:

```
{ /key IncrementCount value }
```

This consists of the current key, the executable name "IncrementCount," and the value of the current key-value pair.

Our *2 index* has us part-way there: we have the mark (the beginning of the eventual procedure) and the key name on the stack.

```
/IncrementCount cvx          % => /key  val  mark  /key  <IncrementCount>
```

Now we place the name *IncrementCount* on the stack and convert it to executable. Note that we had to start with the literal name, so that PostScript would not immediately look up the name and execute the procedure during the construction or our array.

Next Page ->

Acumen Journal: PostScript Tech 24

```
4 -1 roll                          % => /key  mark  /key  <IncrementCount>  val
```

Finally, we bring the value to the top of the stack using *roll,* everyone's favorite stack operator.

```
]                                  % => /key  [/key <IncrementCount> val]
cvx                                % => /key  {/key <IncrementCount> val}
```

We can now create our procedure body. We create the array using the close bracket operator, "]". This leaves on the stack a literal array containing our procedure contents. The call to *cvx* ("convert to exectutable") converts the literal array to an executable procedure body.

```
} if
```

Our *if* clause exits with the key on the stack and, above that, our newly-constructed procedure body. If the *if* block wasn't executed (because this key-value pair did not represent an operator definition), then the stack will now contain the original key-value pair placed on the stack by *forall.*

```
    systemdict 3 1 roll put        % This is going into our systemdict
} bind forall
```

Our *forall* loop procedure ends by putting the two items on the stack (the key and either the procedure body or the original value) into our newly-made *systemdict.*

Our version of *systemdict* now contains all of the PostScript operator names, each associated with a procedure that increments the appropriate value in *profileDict* and then executes the original operator definition.

Next Page ->

Acumen Journal: PostScript Tech 25

Define Utility Procedures

```
/StartProfiler
{    systemdict begin
     //globaldict begin
     //userdict begin
} bind def
```

The *StartProfiler* procedure simply places our *systemdict* on top of the dictionary stack and piles the standard *globaldict* and *userdict* on top of it. From now on, references to PostScript operators will increment the tally in *profileDict.*

```
/StopProfiler
{ countdictstack dictstack0 sub { end } repeat } bind def
```

StopProfiler simply removes everything off the dictionary stack except the three default dictionaries normally there at startup.

```
/ReportCalls
{    profileDict
     {    dup 0 ne
          { exch str cvs print (: ) print = }
          { pop pop }
          ifelse
     } forall
} bind def
```

And, finally, *ReportCalls* steps through all the key-value pairs in *profileDict* and prints to *stdout* any non-zero values.

Next Page ->

Acumen Journal: PostScript Tech 26

Using the Profiler

Our sample code ends with an invocation of *StartProfiler.*

```
% ======== Begin Test ===========
StartProfiler
72 600 moveto 100 100 rlineto 0 -100 rlineto closepath
stroke showpage
StopProfiler
% ======== End Test Subject ===========
ReportCalls
```

We call *StartProfiler,* execute the PostScript code we want to test, call *StopProfiler,* and, finally, execute *ReportCalls,* which dumps the non-zero counts to *stdout.*

Using the Counter

This seems to work reasonably well. Paste everything through the invocation of *StartProfiler* in front of any arbitrary PostScript and calls to *StopProfiler* and *ReportCalls* afterward and it emits an operator count. It seems to work with all the PostScript code I could easily find, rummaging around on my hard disk.

Is it useful?

I don't know exactly how useful this is, from a diagnostic standpoint. Certainly, I think it's a nice demonstration of some programming techniques. I *would* be curious to know if anyone ends up using this to do something akin to real profiling of their PostScript code.

Is it Fun?

Well, yes!

(However, see past, plentiful comments about the simplicity of my social life.)

Return to Main Menu

Schedule of Classes, May – Jul 2003

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	May 19–23		Jul 14–18
New!	**Variable Data PostScript**		Jun 16–20	
	Advanced PostScript		Jun 2–6	
	PostScript for Support Engineers	May 26–30		Jul 28–Aug 1
	Jaws Development		*On-site only*	

PostScript Course Fees PostScript classes cost $2,000 per student.

On-Site Classes These classes may also be taught on your organization's site. Go to www.acumentraining.com/on-site.html for more information.

Registration Info →
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

No Acrobat classes scheduled for this quarter. See the Acumen Training website regarding setting up an on-site class.

Interactive Acrobat

Creating Acrobat Forms

Troubleshooting with Enfocus' PitStop

Acrobat Class Fees

Acrobat Essentials and *Creating Acrobat Forms* ($\frac{1}{2}$-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

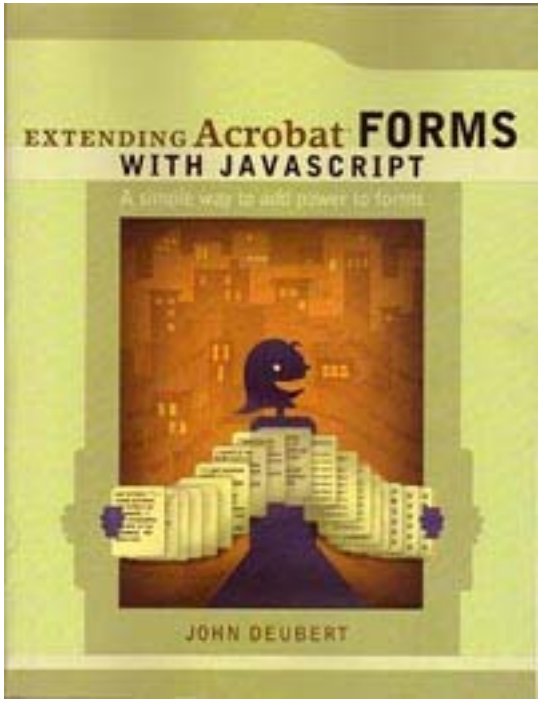
Back issues of the Acumen Journal are available at the Acumen Training website: www.acumenjournal.com/AcumenJournal.html

Return to First Page

What's New at Acumen Training?

Hooray! A New Book

I have a new book out: *Extending Acrobat Forms With JavaScript.* This book is a non-programmer's guide to adding features to your Acrobat forms using JavaScript. If you are a form designer with good Acrobat experience, but have never written a line of JavaScript, C, or other programming code before (and were pretty sure you never wanted to do so), then this book is for you.



We introduce programming concepts as we learn how to add specific enhancements to your forms; for example, we talk about arrays while adding a price table to a form, case statements while creating a pop-up menu, if…else commands while checking the version of the user's Acrobat viewer.

Extending Acrobat Forms is available from Amazon.com and bookstores everywhere.

For a list of chapters and other information, go to:

www.acumentraining.com/Book-AcroJS.html

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? Does it make you acutely aware of your tongue?

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