

Table of Contents

[The Acrobat User](#)

Rollover Help

Acrobat supplies automatic support for "tool-tip" help for form fields. A common alternative to tool-tip help is "roll-over" help that appears as soon as the mouse moves over a control. This month we'll see one way of implementing this in Acrobat.

[PostScript Tech](#)

Text Along an Arc

Something fun this month: we'll look at one method of printing text along an arc. Along the way, we'll discuss a bit about the nature of PostScript procedures.

[Class Schedule](#)

Nov-Dec-Jan

Where and when are we teaching our Acrobat and PostScript classes? See [here](#)!

[What's New?](#)

New Project

Another book project coming up.

[Contacting Acumen](#)

Telephone number, email address, postal address, all the ways of getting to Acumen.

[Journal feedback: suggestions for articles, questions, etc.](#)

Implementing Roll-Over Help

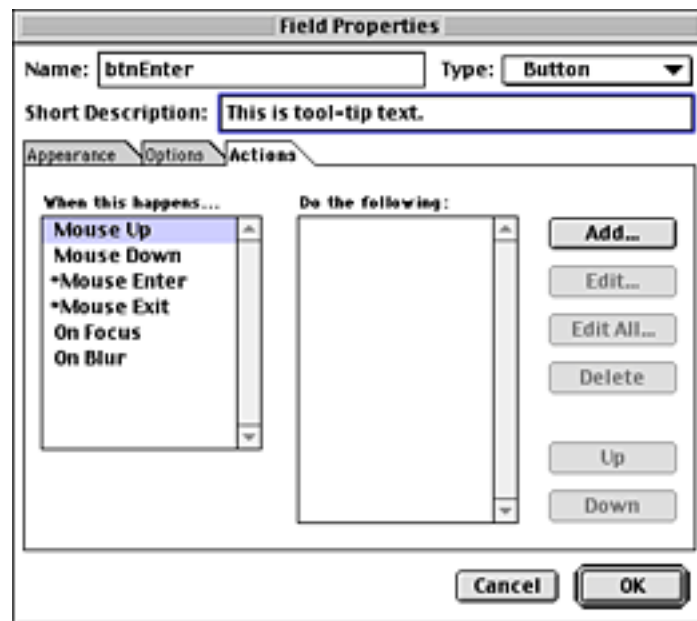
Consider the two buttons at right. Move your mouse cursor over the first one and leave it there for half a second.
(This is not rhetorical: do it, please!)

After a moment, you will get a little text box that appears with some tool-tip text. Acrobat supplies this feature automatically. In the Field Properties dialog box, there is a text field labeled "Short Description." Anything you enter here will be presented to the user if the cursor rests over the form field for a half second. This is what I use to label the navigation buttons at the top of each *Journal* page.

Now move your mouse over the second button. As soon as the cursor enters the control, a window appears with help text in it. This is "roll-over" text, appearing as soon as the cursor rolls over the control.

Unfortunately, Acrobat does not provide automatic support for roll-over help; we need to do it ourselves with a combination of "Mouse Enter" and "Mouse Exit" actions.

Let's see how.



[Next Page ->](#)

Our Example

In this article, we shall be adding roll-over help to the “Enter” button in the PDF file pictured at right. As the cursor moves over the button, some helpful text will appear, as in the far right illustration.

Overview

The trick is that there are actually two form fields in the *Axolotlville* PDF file:

- The *Enter* button.
- A text field, initially invisible, holding the help text.

We create the text field in a hidden state. The *Enter* button has a *Mouse Enter* action that makes the help field visible and a *Mouse Exit* action that hides it again.

Note: In the discussion that follows, I assume you have had at least some experience making Acrobat form fields using the *Form* tool.



[Next Page ->](#)

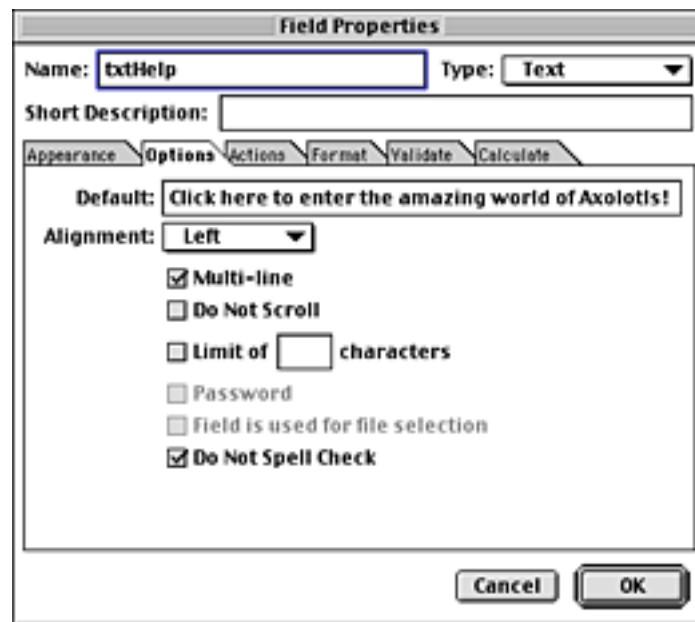
Making the Text Field Our discussion assumes you have already created the *Enter* button. We shall add the roll-over help to the page.

1. *Create the text field* Selecting the Form tool, click and drag out the form field that will be our tool tip text.



Acrobat will present you with the Field Properties dialog box, at right.

2. *Assign a Type and Name* Select *Text* in the *Type* menu and then give your text field a name (in the *Name* field). The name can be anything you like, though I favor short, descriptive names. In the illustration at right, I named my text field "txtHelp."



The image shows the 'Field Properties' dialog box with the 'Options' tab selected. The 'Name' field contains 'txtHelp' and the 'Type' is set to 'Text'. The 'Short Description' field contains the text 'Click here to enter the amazing world of Axolotis!'. The 'Alignment' is set to 'Left'. The 'Multi-line' checkbox is checked, while 'Do Not Scroll', 'Limit of characters', 'Password', and 'Field is used for file selection' are unchecked. The 'Do Not Spell Check' checkbox is checked. The 'Default' text field also contains the same text. The 'Cancel' and 'OK' buttons are at the bottom right.

3. *Options Panel* Do the following in the *Options* pane:

- Type the text you want for your roll-over help into the *Default* text field.
- Select the *Multi-line* checkbox.

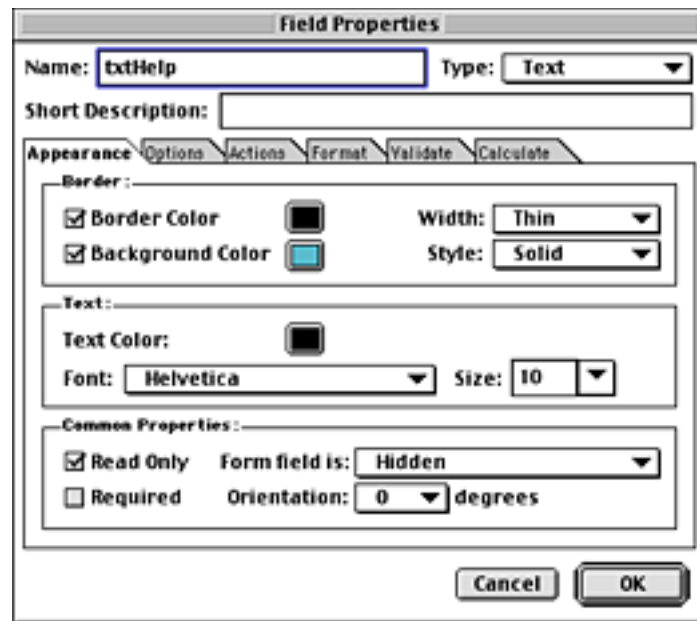
I recommend you do this even if you have only a short snippet of text. It won't hurt in that case and prevents surprises later if you modify the help text.

[Next Page ->](#)

3. *Appearance Panel* Go to the *Appearance* panel and do the following:

- Set the Border and Background Colors to whatever you like. I like pastel colors for the background of help text, but feel free to indulge your own tastes.
- Set the Border Width to *Thin* and the Border Style to *Solid*.
- Set the *Form field is:* menu to "Hidden."
- Select the *Read Only* checkbox.

Setting the *Read Only* property is optional, since the user shouldn't ever be able to click on the help text field. (It disappears when they move the mouse off the *Enter* button.)



4. *Click OK* This will dismiss the *Field Properties* dialog box, returning you to your PDF page.

You are now looking at your *Axolotl* page again. So far, the changes you have made are visually boring. (Looking at invisible objects is always a bit unexciting.)

Now let's add the trigger that lets us use this text field as roll-over help.

[Next Page ->](#)

Setting the Actions We need to add two actions to our *Enter* button:

- A *Mouse Enter* action that makes our help text visible when the mouse rolls onto the button.
- A *Mouse Exit* action that hides our text again when the mouse rolls off the button.

We will implement these activities with *Show/Hide Field* actions attached to the appropriate events for our button.

1. Go to "Field Properties" With the Form Tool selected, double-click on the *Enter* button. You will be faced with the Field Properties for the button.



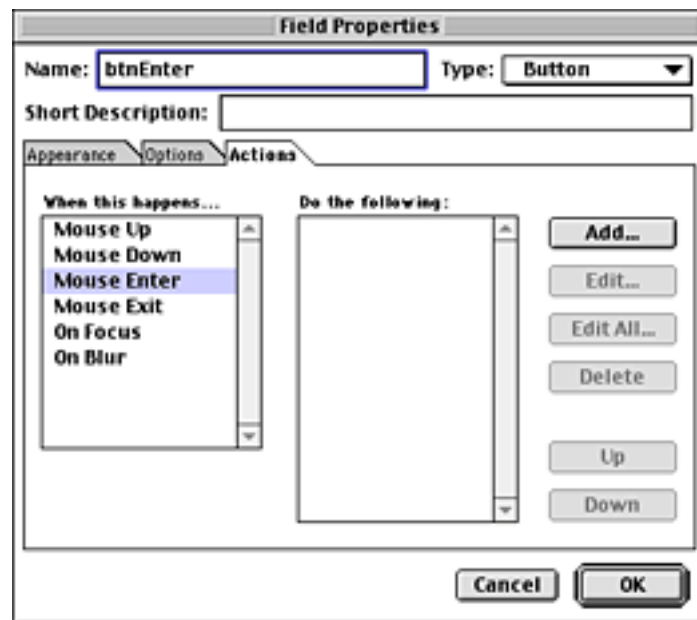
We are interested in the *Actions* pane of this dialog box.

Here, we are going to attach an action to each of the two events of interest.

2. Add the *Mouse Enter* Action

Click on the *Mouse Enter* event and then on the *Add...* button.

Acrobat will present you with the *Add an Action* dialog box, pictured on the next page.



[Next Page ->](#)

3. *Add Show/Hide Field* In the *Add an Action* dialog box, do the following:

- Select *Show/Hide Field* from the *Type* Pop-up menu.
- Click on the *Edit* button.

Acrobat will present you with the *Show/Hide Field* dialog box (below, right).

- Select your help text field in the list (*txtHelp* in the illustration) and click on the *Show* button.

Since this is the *Mouse Enter* event, we want to make the help text visible.

- Click *OK* in the *Show/Hide Field* dialog box and *Set Action* in the *Add an Action* dialog box.

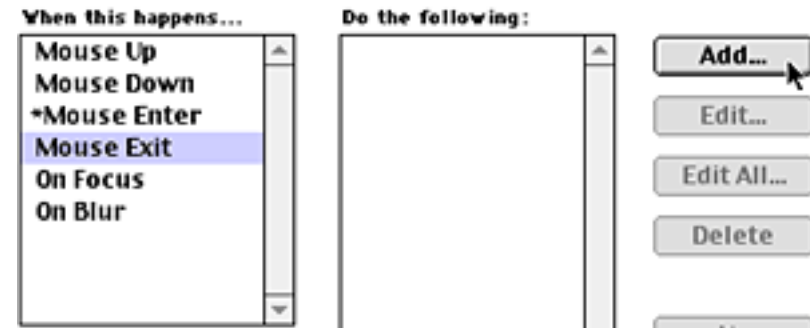
This returns us to *Field Properties*.



[Next Page ->](#)

4. *Add the Mouse Exit Action* Now let's add the *Mouse Exit* action in the same way:

- Click on the *Mouse Exit* event and then click the *Add...* button.
- In the resulting *Add an Action* dialog box, select *Show/Hide Field* in the pop-up menu and click on the *Edit* button.
- In the *Show/Hide Field* dialog box, select your help text field and click on the *Hide* button.
- Exit out of all dialog boxes until you are looking at your PDF page again.



Done! That's all there is to it. We now have fully functioning roll-over help attached to our button.

Note that you can place invisible buttons (that is, with no border or background) over anything on your PDF page, letting you attach roll-over help to [text](#), graphics, or anything else you choose.

[Next Page ->](#)

JavaScript Help

Our method of creating roll-over help using *Show/Hide Field* actions works very well if you have only a small number of buttons on a page for which you need to supply help. It becomes unwieldy if you have a large number of help fields; managing all those invisible text fields becomes messy.

A better strategy in that case is to use a single help text field and use a JavaScript for your *Mouse Enter/Exit* actions that puts help text into the text field and makes the field visible. Unfortunately, this technique is more than we have room for this month.

Next month, perhaps.

The screenshot shows a registration form for 'Acumen Training - PostScript & Acrobat Training'. The form includes fields for Name, Company, Address, Billing Address, P.O. Box, Telephone, Fax, E-Mail, and Signature. A diagram overlays the form, showing a network of hidden text fields (e.g., 'btHelpName', 'btHelpCompany', 'btHelpAddress') connected by lines, illustrating the 'Show/Hide Field' action technique. A red box highlights the 'btHelpSchro' field. The form also features a 'Submit' button and a 'Go Back' button.

[Return to Main Menu](#)

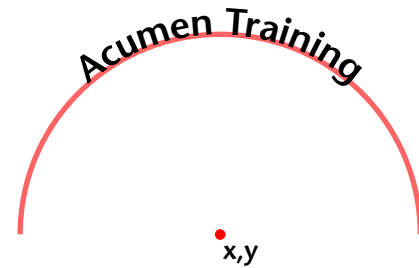
Printing Text Along an Arc

This month we'll look at a PostScript snippet that prints arbitrary text along a circular arc. We shall define a procedure called *arcshow* that takes a string and a radius from the operand stack and prints the string on an arc centered on the currentpoint.

That is, the following PostScript code would yield the text diagrammed at right.

```
x y moveto  
(Acumen Training) 50 arcshow
```

Along the way, this little piece of code will let us discuss some unobvious points about PostScript procedure bodies.



[Next Page ->](#)

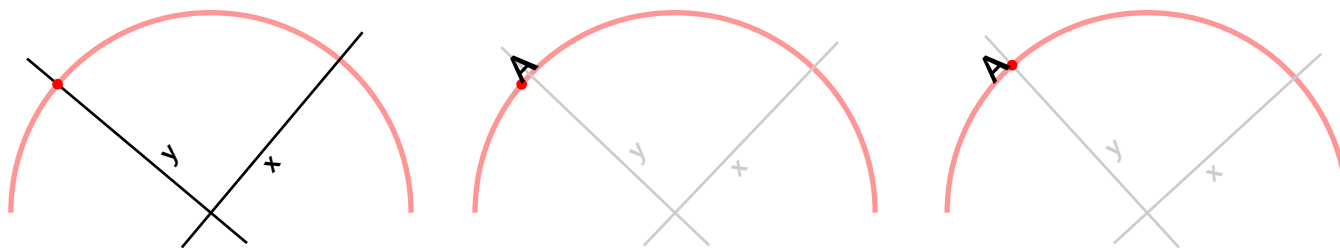
The Algorithm

Here's how we're going to print our text:

1. Translate to the current point.
2. Calculate how much of an arc our string will subtend and rotate counterclockwise by half that amount.

The first character will print where the y axis now intersects the arc.

3. Do the following for each character in the string:
 - Do a *moveto* to a point r units up the y axis (where r is the radius of the circular arc.)
 - Rotate by half the angle subtended by the character we're printing.
 - Print the character.
 - Again rotate by half the angle subtended by the character. This leaves us ready to print the next character.



[Next Page ->](#)

The Code

Here's the PostScript. It extends across two pages; sorry.

As usual, this PostScript code is available on the [Acumen Training Resources page](#). Look for the file *arcshow.ps*.

```
/char 1 string def

% Given a string length, return the angle subtended by that string.
% (360OverCircumference is calculated at runtime.)
/arcLenToAngle      % stack: strlen => angle
{ 360OverCircumference mul } bind def

/_arcshow           % (c) => ---
{
    dup stringwidth pop arcLenToAngle      % (c) angw
    -2 div dup rotate                      % (c) -angw/2
    exch show                               % angw/2
    rotate                                  % ---
} bind def
```

[Next Page ->](#)

```
/arcshow          % (str) radius => ---
{
    gsave
    currentpoint translate          % Move origin to currentpoint
    dup 6.28 mul 360 exch div        % Calculate 360OverCircumference
    /360OverCircumference exch def   % ( = 360/2πr)
    exch dup stringwidth pop 2 div   % Rotate counterclockwise by half
    arcLenToAngle rotate             % the angle subtended by str
    0 3 -1 roll moveto              % Move to the (0,radius).
    {                                % Do a forall with the string
        char 0 3 -1 roll put        % Put each charcode into char
        char _arcshow               % Print it along the arc
    } forall
    grestore
} bind def

% Now let's use it:
/Helvetica 12 selectfont
300 400 moveto
(Acumen Training) 50 arcshow

showpage
```

[Next Page ->](#)

Stepping Through the Code

Let's look at our program a piece at a time.

Some initial definitions

```
/char 1 string def
```

```
/arcLenToAngle      % stack: strlen => angle  
{ 360OverCircumference mul } bind def
```

We start by defining a 1-character string, *char*. We shall use this to convert a character code to a printable string (e.g., convert the character code 65 into the one-character string (A)).

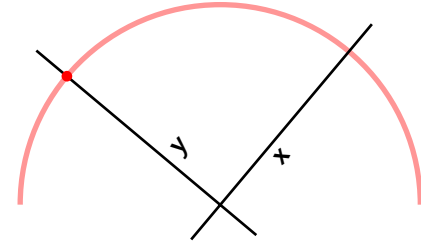
We also define a procedure that calculates the angular measure subtended by a given string length. Remembering our trigonometry, we calculate this by:

$$\text{angle} = 360 \times \text{strlen} / \text{circumference}$$

Our definition of *arcLenToAngle* uses a variable, *360OverCircumference*, which we shall need to calculate before we use the procedure.

[Next Page ->](#)

`_arcshow` The `_arcshow` procedure prints a one-character string on our arc; we shall call this once for each character in our string. The procedure assumes that the coordinate system has been rotated so that the *y* axis intersects our arc at the current point, where our character is to be printed.



```
/_arcshow      % (c) => ---
{
    dup stringwidth pop arcLenToAngle      % (c) angw
    -2 div dup rotate                      % (c) -angw/2
    exch show                               % -angw/2
    rotate                                  % ---
} bind def
```

In detail, here's what's happening:

```
dup stringwidth pop arcLenToAngle      % (c) angw
```

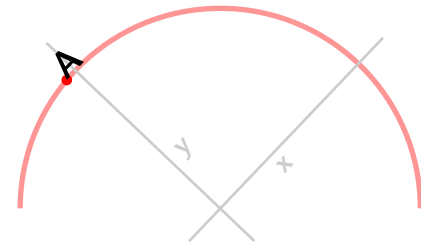
We duplicate the string, calculate its width, and then calculate the angular measure that corresponds to that string width.

```
-2 div dup rotate                      % (c) -angw/2
```

We halve that angle (and reverse its sign) and then rotate clockwise by that amount.

The character, when we print it, will now be centered about the *y* axis and will be oriented properly for the piece of the circular arc that it occupies.

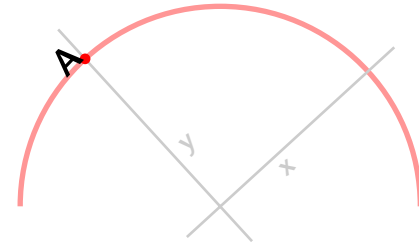
[Next Page ->](#)



```
exch show
rotate
```

```
% -angw/2
% ---
```

Finally, we print the string and then rotate the other half of the character's angular width. This leaves the *y* axis running through the current point's new location (the latter having been moved by the *show* operator).



arcshow This is the procedure we directly call in our PostScript code. It takes a string and a radius from the operand stack and prints that string along an arc. The arc is centered on the current point and has the specified radius.

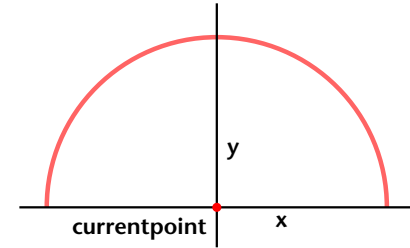
```
/arcshow % (str) radius => ---
{
    gsave
    currentpoint translate
    dup 6.28 mul 360 exch div
    /360overCircumference exch def
    exch dup stringwidth pop 2 div
    arcLenToAngle rotate
    0 3 -1 roll moveto
    { char 0 3 -1 roll put
      char _arcshow
    } forall
    grestore
} bind def
```

```
% (str) radius
% radius (str) strwid/2
% radius (str)
% (str)
```

[Next Page ->](#)

In Detail... `gsave
currentpoint translate`

We save the graphics state and move the origin to the current point.



```
dup 6.28 mul 360 exch div  
/360OverCircumference exch def      % (str) radius
```

We define the variable *360OverCircumference*. (You *do* remember that the circumference of a circle is $2\pi r$, don't you?)

Note that our procedure's arguments still remain on the operand stack.

```
exch dup stringwidth pop 2 div      % radius (str) strwid/2
```

We calculate the width of our string and then halve that width.

```
arcLenToAngle rotate                 % radius (str)
```

We convert the half-width to an angular measure and then rotate that many degrees counterclockwise.

```
0 3 -1 roll moveto                  % (str)
```

The last thing we do before we fire up our loop is *moveto* to a position the radius' distance up the y axis. (We put a *0* on the stack—our x coordinate—and then do a *3 -1 roll*, bringing the radius to the top of the stack.

[Next Page ->](#)

```
{ char 0 3 -1 roll put
  char _arcshow
} forall
```

Finally, we do a *forall* loop that steps through each character code in the string (the string still being on the stack). Each time it is executed, the *forall* procedure will find on the stack an integer representing a character code from the string.

The procedure puts the character code into position 0 of the *char* string. (Since *char* is a one-character string, position 0 is the only position it has.) It then hands *char* to the *_arcshow* procedure, printing the character at the current position on our arc.

grestore

Finally, we restore the graphics state to its previous condition. Note that *arcshow* has no net effect on the current point, which is left at the center of the arc.

Use the procedure Finally, we print some text using our new *arcshow* procedure.

```
/Helvetica 12 selectfont
300 400 moveto
(Acumen Training) 50 arcshow
```

Note that after our call to *arcshow*, the current point is still at 300,400.

[Next Page ->](#)

Weaknesses This isn't the only way to print text along an arc, of course. I'm sure one could make a case for other, better methods. The main weakness of this technique is that it only draws text along the upper arc.

For extra credit, modify *arcshow* so it will print text centered about the lower arc, as at right. You'll need to devise some way to let *arcshow* know which arc you want. (Maybe a negative radius could signify a bottom arc?)

Acumen Training

Improvements (Well, sort of...)

Let's make a couple of changes to our PostScript program. One of these changes is an improvement. The second change makes your PostScript code marginally more efficient while rendering it indecipherable by most PostScript programmers; but it's *fun*.

Pretty fun, anyway.

[Next Page ->](#)

The improved code Here's how our program looks with these changes:

```
/arcLenToAngle          % len => angle
{ 1 mul } bind def      % The 1 is a place holder

/_arcshow               % (c) => --- % No changes to _arcshow
{ dup stringwidth pop arcLenToAngle      % (c) r ang
  -2 div dup rotate                      % (c) r ang
  exch show rotate
} bind def

/arcshow % (str) radius => ---
{ gsave
  currentpoint translate
  /arcLenToAngle load      % Get the arcLenToAngle procedure body
    0                      % index 0 into the procedure body
    2 index 6.28 mul 360 exch div % calculate 360 / 2πr
    put                    % put value into proc. body, replacing "1"
  exch dup stringwidth pop 2 div
  arcLenToAngle rotate
  0 3 -1 roll moveto
  { (X) dup 0 4 -1 roll put _arcshow % Here using an in-line string
  } forall                          % constant instead of char
  grestore
} bind def
```

So, what's better?

[Next Page ->](#)

Eliminate */char* The first change we made was to eliminate the one-character string, *char*. Instead of having a named string constant into which we put our character codes, one at a time, we shall simply place a string constant in-line into our *forall* procedure.

```
{ (X) dup 0 4 -1 roll put _arcshow } forall
```

The single character in *(X)* is simply a place holder; the *put* replaces that character with the character code currently being processed in the loop.

By the way, this looks though it would be a memory leak; it's not. Strings are created at scanning/tokenizing time, when the procedure body is constructed. The parentheses tell the tokenizer to create a string object, containing a pointer to some place in VM; it is this object-mit-pointer that is placed into the *forall*'s procedure body.

[Next Page ->](#)

Eliminate *360OverCircumference*

We also managed to get rid of the variable *360OverCircumference*. This took a little more work, since the value of this variable is calculated at runtime. To do this, we take advantage of a little known fact regarding the nature of procedure bodies.

A procedure body is really an array; the array object is marked *executable* (which is what makes it a procedure body), but it is otherwise exactly like any other PostScript array. In particular, you can use *put* to alter the contents of a procedure body.

We make use of this fact to set *at runtime* the constant value that goes into the *arcLenToAngle* procedure.

In our PostScript code, we now define *arcLenToAngle* as follows:

```
/arcLenToAngle { 1 mul } bind def
```

The *1* in this procedure body is purely a place holder. The *arcshow* procedure replaces it at runtime with the actual value, which we originally associated with the name *360OverCircumference*.

```
/arcLenToAngle load      % { 1 mul }  
    0                    % { 1 mul }  0  
    2 index 6.28 mul 360 exch div      % { 1 mul }  0  360/2 $\pi$   
    put                          % put value into proc. body, replacing "1"
```

The procedure body associated with *arcLenToAngle* ends up containing the numeric value $360/2\pi r$ and the executable name *mul*.

[Next Page ->](#)

However... I can't say this second "improvement" is particularly worth doing, but it is an unusual programming technique in PostScript and I thought it would be fun to show it off.

Truth be told, I don't really recommend doing this; the miniscule savings in time you gain (from eliminating a name lookup) isn't worth the loss in readability.

It *is* fun, though.

[Return to Main Menu](#)

Schedule of Classes, Nov 2002 – Jan 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](#)

December 2 – 6

[Advanced PostScript](#)

November 11 – 15

[PostScript for Support Engineers](#)

January 20 – 24, 2003

[Jaws Development](#)

On-site only; see the Acumen Training website for more information.

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](#)

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 (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?

New Project Brewing

I've no particular information to pass out just now, but I'm starting work on a new project.

Tell you about it next month, probably.



Creating Acrobat Forms
John Deubert, Adobe Press

*"I found it immensely helpful
in settling the Gauls' hash."
— J. Caesar*

[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 inspire you try to conduct your own appendectomy?

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](#)