Some PostScript and Acrobat .PDF Post Document Editing Tools

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

http://www.tinaja.com
don@tinaja.com
(928) 428-4073

There are times and places where you might want to modify several existing **PostScript** or **Acrobat** documents. Perhaps the files are way too long because of inefficient machine generation or translation.

Or you want to add borders or logos or IP statements. Or you might wish to programmatically extract form data for a different reuse. Or you need to reformat text for an entirely different application. Or want to resize or reorient. Or want to make major changes to the final document use or appearance.

In general…

> **The Acrobat .PDF format is clearly NOT intended for simple or easy post-document editing!**
>
> **If at all possible, GO BACK TO THE ORIGINAL DOCUMENT SOURCECODE for any and all edits or data capture.**
>
> **Use the tools and techniques shown here ONLY if and when no reasonable alternatives exist.**

There is no particular sequence with which any marks will appear on an **Adobe Acrobat** page. What may seem like a block of text might be put down as its individual words or even individual letters. It can often be enormously difficult to, say, edit a paragraph where words will reflow between lines. Or to distinguish between words in a figure and words in a text body.

For the tools and techniques we are about to look at to work, you will have to be a "bare metal" machine language type of person. Access and **thorough** familiarity with both the **PostScript Language Reference Manual** and the **PDF Reference** are a must. As is understanding exactly how to go about **Using Acrobat Distiller as a General Purpose PostScript Computer**.

Our approach here will be based upon…

.PDF Acrobat editing is often best done by…

(1) Print to disk to recapture an equivalent to the underlying PostScript code as a clean textfile.

(2) Thoroughly study the captured code to understand exactly what happens in which sequence.

(3) Edit or modify the captured PostScript code.

(4) Redistill the modified ps code back to Acrobat.

A **print to disk** function is usually built into most **Acrobat** or **GhostScript** print drivers. This will let you convert from Acrobat back down to raw PostScript. Usually a non-obvious box will have to be clicked to switch from paper to file.

There are many print to disk options…

When doing a .PDF print to disk…

- **Turn off ALL compression.**
- **Print to Level II compatibility.**
- **Print as 7-bit clean ASCII text.**

Getting a **7-bit clean ASCII textfile** will usually be an essential starting point to your editing process.

There are some possible complications as your images may get extremely long in **ASCII85** format. And treatment of **transparency** also may end up tricky because this newer .PDF feature has no direct PostScript equivalent. Should any of these complications arise, just **bypass them temporarily and reintroduce them later**.

Let's turn to some individual tools and techniques that you might find most useful for .PDF post editing. Chances are you may want to mix and match these so they are appropriate to your particular needs…

Working With Disk Files

One of the most powerful (and least appreciated) features of the **PostScript** language is its superb ability to **write or read most any diskfile in most any format and most any language**. It is also possible to convert most ordinary **PostScript** strings into a file that can be read from or written to. But subject to a strict maximum 65K size limit.

If your target doc has several pages, **it may be best to split up the doc into a new header file, new individual page files, and a new footer file**.

This can end up quite useful if you want to edit many files **programmatically**, rather than one file **manually**. Because footers are not usually a big deal, you can sometimes simply leave them tacked onto the last page. These split files are easily recombined later.

There are some specific rules for PostScript access to filenames in a **Windows XP** or similar environment…

- **The FULL file pathname MUST be used.**
- **Spaces are discouraged in directory or subdirectory filenames. Use split_name instead.**
- **DOUBLE REVERSE SLASHES are required INSIDE any PostScript string that needs a single reverse slash.**

Here is how you create a proper filename for PostScript access. First, you may need this **/mergestr** routine from our **Gonzo** utilities…

```
% /mergestr merges the two stack strings into one...

/mergestr {2 copy length exch length add string dup dup
4 3 roll 4 index length exch putinterval 3 1 roll exch 0 exch
putinterval} def
```

Here is how you might prepare a pair of host based diskfiles for reading and writing…

```
/sourcefileprefix
  (C:\\Documents and Settings\\don\\Desktop\\ps_story\\)
   store
/sourcefilename (demosource1.psl) store
/destfilename  (demodest1.psl) store

/s4file sourcefileprefix sourcefilename       % source file
mergestr (r) file store

/d4file sourcefileprefix destfilename         % destination file
mergestr (w) file store
```

Other options are to use **(w+)** to read and write. Or **(a)** to append.

The usual problem with an **-invalidaccess-** error message is having the wrong directory and subdirectory sequence. Or not properly merging it. In interest of sanity, **it pays to keep all read and written files in the same subdirectory**.

Some further details: Normally, you would read or write a file in sequential order. But you may want to read and rewrite part of a file up to a marker, insert some new stuff, and then continue on. It turns out there are **PostScript** commands of **-fileposition-** and **-setfileposition-**.

These can be used to find where something happens in a file or set the file to that location for reading or writing. One quirk: **-setfileposition-** "expires" if you ever read an entire file. Or try to write beyond the end of an existing file. If either happens, **you will have to close and reopen the file for any further read or write access**.

There is no direct way to find the length of a disk based **PostScript** file. Instead, you have to combine **-fileposition-** and **-bytesavailable-** …

To find the size of a PostScript file:

... file dup fileposition exch bytesavailable add ...

Three File Options

There are three different **PostScript** commands for reading a host disk based file. These are **-read-**, **-readline-**, and **-readstring-**. Each has particular strengths and weaknesses.

With **-read-**and its companion **-write-**, a file can be read **one byte at a time**. And returning an **0-255** integer byte. An ASCII upper case "**A**" thus might be decimal **63**. This method is often the best for complex disk manipulations in that there are no issues with end-of-line characters or any **string dereferencing** hassles.

The PostScript buffering internals are swift enough to only actually do disk reads or writes in larger blocks. So, this **-read-** method is reasonably fast and does not create excessive disk wear.

With **-readline-**, one line of the input file is read to a string, **but excluding any end-of-file markers**. Your work string has to be longer than the longest line to be read, but **1024** bytes is often more than enough. The returned strings can be searched for markers to be edited. If the edited strings are to be rewritten to disk, end-of-line markers (such as **(\n)**) will have to be inserted between string writes.

With **- readstring-**, a workstring gets **completely** filled with everything that will fit from the input file. Including any and all end-of-line characters. As with most any **PostScript** string, **there is a 65,535 byte maximum length**.

If you are going to break up a larger input file into pieces, I feel it is best to first use **-readline-** to scan and then search the document for break points. A **hitlist** array of break filepositions can then be created. Finally, the blocks between the marking file positions can be transferred using **-read-** and **-write**.

We'll see some example code shortly.

Strings as files?

Yes, it is possible to convert most any **PostScript** string into a file that can be read or written. Here is how to read a string the same as a disk file…

```
/str1 (This is a test) store
/file1 str1 0 (%EOF) /SubFileDecode filter store
6 {file1 read pop == } repeat
```

Here our original string is **str1**. To create **file1**, three parameters of the original string, an occurance counter, and an end of string marker are passed through a **SubFileDecode** filter. Our final line reads the first six ASCII characters as equivalent integers 84, 104, 105, 115, 32, and 105.

Creating a write file is equally simple…

```
/str2 (zzzzzzzzzzzzzzzzz) store
/file2 str2 /NullEncode filter store
file2 65 write file2 66 write file2 67 write str2 ==
```

This time, we use a file write to replace the first three string characters with **(ABC)** .

Remember there is a strict **65K length limit** on any string file read or write.

Here is a mind-bogglingly obscure example use for strings as files. Where we do an ultra sneaky **numeric array to string conversion**…

```
/makestring {dup length string dup /NullEncode filter
3 -1 roll {1 index exch write} forall pop } def
[72 105 32 116 104 101 114 101 33] --> (Hi there!)
```

Similar sneaky tricks and useful stunts can be found **here**

Splitting up and modifying an Input File

As we've seen, it often can be useful to split up your task into individual header and page files. Doing this manually is no big deal. Just use simple cut and pastes. Visually keying on obvious divisions.

Programmatically splitting up a file can be a tad trickier. And is often best done using the file techniques we just looked at. Exactly **how** you split your input file will be highly dependent on the third party code used to create it. A careful study of your input code will be needed to determine suitable break points.

Very often, there will be a **"%%Page:** marker after the header and immediately before each page. Once again, I feel it is best to scan the document for markers and create a **hitlist** of file positions. That hitlist array can then be used to create the individual files.

Here is how you might scan for a **"%%Page:** or other marker and create a hit list from it…

```
/prevfileposn 0 store                    % previous file position
/hitlist mark 0                          % start hitlist array

20000 { s1file ws1 readline
        { (%%Page:) search                % look for page starts
        {pop pop pop prevfileposn}        % mark start if found
        {pop /prevfileposn
        s1file fileposition store} ifelse % save old position
        } {pop exit} ifelse} repeat

        prevfileposn                      % mark file end
          ] store                         % save hitlist
```

Your hitlist will then be an array holding the initial starting file position for each new file to be created and saved to disk.

Here's how to get from a pagecount index (such as **jj**) to a new filename…

```
/d1file sourcefileprefix (~page)
mergestr jj 10 string cvs mergestr
(.psl) mergestr (w) file store
```

Finally here's how you use two hitlist entries to create a new file…

```
hitlist jj get 1 hitlist jj 1 add get 1 sub
  {s1file exch setfileposition
   s1file read {d1file exch write} if} for
```

Note the **1 sub**. Normally, your new page file should end **one file position count before** the next one starts.

Changing Page Headers

You likely will want to modify your split out page files. Perhaps by changing their internal header, adding new post-header code to do your stuff, and finally the page content itself.

A captured single page doc of **~page8.psI** might be resaved as **~page8m.pdf**.

Each page should have some defining point where its header ends and its live content begins. These markers will be highly third party sourcecode specific. In one recent client example, the final header code lines were a unique **end end**. To rewrite the unchanged header, you would read **up to** the fileposition just past the **end end** marker and write this to your modified file.

If you are to insert some new post header code, you would do so at this point. You then would continue with the actual page content on your original file and write it to your new modified file. Starting with your saved file position.

Here is how you programmatically insert new post header page code…

```
   d2file (\n\n%% BeginInsertedMods\n\n) writestring

   d2file (\nnew ps code line for everybody #1) writestring
   d2file (\nnew ps code line for everybody #2) writestring
   d2file (\n  . . . etc . . . #x) writestring

jj 3 eq {
   d2file (\nnew ps code line for page3 only #1) writestring
   d2file (\nnew ps code line for page3 only #2) writestring
   d2file (\n  . . . etc . . . #x) writestring
      } if

   d2file (\n\n%% EndInsertedMods\n\n) writestring
```

Your code can be made conditional on individual pages as shown above.

An alternate page header might be needed if you are changing page sizes, are switching to or from landscape, or need similar mods. To handle this, you would

start off reading and writing an alternate header file. Then you add your post header code. Finally, you would go back to the **original** page header end file position marker and write the content specific page info.

A suitable flag could be used to conditionally select the original or alternate header.

Recombining Files

After your rework and modifications, you may want to recombine your individual page files back into a single document. A doc you can send to Acrobat Distiller to create your final output .PDF file.

The simplest way to recombine files is to write a ps scripting program that runs one modified header or one new page at a time…

```
(~headerm.psl) run
(~page1m.psl) run
(~page2m.psl) run
  .  .  .  .  .  .
(~pagexm.psl) run
```

Otherwise, a read-write loop can be done to combine your modified files. A run script is obviously faster and more convenient. But it requires the presence of all the individual files every time it is used.

The Crucial Key to Postdoc .PDF Editing

The key secret tool to Acrobat post editing is…

To Post-Edit an Acrobat .PDF or PS file...

(1) Begin a new unlocked and predefined dictionary.

(2) Redefine -showpage- and similar procs so they do a combination of old and new tasks.

(3) Be sure to use -def- and not -store- when making any new definitions.

(4) Be sure to close the new dictionary before any end-of-page -restore-.

Here is an example of a showpage redefinition…

```
/showpage {
        % procs that go ON TOP OF or AFTER
        % existing marks ON THIS PAGE go here.

        systemdict /showpage get exec   % do "real" showpage

        % procs that go UNDERNEATH or BEFORE
        % existing marks ON THE NEXT page go here.

             } def
```

One more time: anything happening **before** the "real" showpage will go **on top of or after** the **present** page marks. Anything **after** the "real" showpage will go **underneath** or **before** the **next** page marks.

An extra print-defeated showpage at document start or ignoring anything extra at document end may be needed to get everything to come out even.

It is also very important to note the procedure in separating "old" and "new" identical commands. Otherwise you may end up calling yourself in an infinite loop. In the case of showpage, a **systemdict /showpage get exec** should work.

In the case of a third party routine buried in an obscure dictionary, you could find the dictionary name and use the above **dictname /procname get exec**. Or simply copy the third party routine into your new top-of-stack dictionary.

Yet another detail: Note that **-def-** will define a proc to the **top** dictionary on the stack. Either creating a new proc or overwriting an existing one. By contrast, any **-store-** will reach down into the dictionary stack as far as it has to when overwriting an existing def or store.

The problem of having to close your new dictionary before a restore can often be done as the last step in any **-showpage-** redefinition. Note that a **-restore-** will generate an error if there is a new dictionary still active.

The original third party software may have **-showpage-** buried in some other proc. Just redefine their own redefinition as needed.

Dealing with -readonly-

Sometimes the third party code will lock out any attempt at creating a new dictionary such as the **/mydict 100 dict def** that you will need sometime before any actual **mydict begin** used before any redefinition.

If this happens, the first thing to try is to **attempt defining your new dictionary very early in the header**. If this does not work, then you will have to modify the existing headers to bypass or else comment out any **currentdict readonly pop** code sequence.

Capturing Text

Very often, you would like to capture actual text. Perhaps for reformatting or editing. Or else to extract form or customer specific data. I have sometimes found it convenient to create a **text data array** of form…

```
/textdata [
        [ xpos0 ypos0 (string0) ]
        [ xpos1 ypos1 (string1) ]
        [ xpos2 ypos2 (string2) ]
            .   .   .   .   .   .   .
        [ xposx yposx (stringx) ]
          ] store
```

Such an array can simply be sent into your log file with a == command. Or else selectively written to a new disk file using our above techniques.

Here is an example of diverting the -**show**- command so it also reports…

```
/show { /curstr exch store              % grab string & x,y
      currentpoint /cury exch store
      /curs exch store

     ([ ) curx 10 string cvs mergestr  % report data
     ( ) mergestr cury 10 string cvs mergestr
     curstr mergestr (]\n)mergestr  ==

     curstr userdict /show get exec   % do actual show
        } store
```

And here is how the somewhat more obtuse -**xshow**- can also report…

```
/xshow { /curxdat exch store          % save x data array
       /curstr exch store             % save current string
       currentpoint /cury exch store  % save currentpoint
       /curx exch store

      ([ ) curx 10 string cvs mergestr % report data
      ( ) mergestr cury 10 string cvs mergestr
      curstr mergestr (]\n) mergestr ==

      curstr curxdat userdict /xshow  % do actual show
      get exec } store
```

One more small detail: The original code may use **clipping windows** to prevent a long data entry from trashing the rest of the form or page. This may prevent the immediate relocation of the entered data, giving you a blank area instead.

One way of dealing with this is to selectively defeat the **-clip-** command and allow overwrites to happen. Otherwise, creative use of the **-clippath-** operator can be used to extract the intended size limits and then suitably handle them.

Another detail: the simple text capture shown here may not allow for special characters, space kernings, or a fill justify. Much of this is easily dealt with using our upcoming **Gonzo** techniques. What remains can often be ignored or custom patched. Otherwise, the actual **curxdat** kerning info can be extracted and reused with some considerably more complicated redefinition procs.

Capturing Data

Once you have a newly saved array of positions and strings, it is a simple matter to reach into that array to selectively extract data.

Ferinstance, **[120.0 75.0 (Arron A. Aardvark)]** might be a customer name. And all similar customer names are likely to be presented at the same **120, 75** position on other pages. You can search on the first two array values for a customer name hit.

Just be sure you **never compare a real against an integer!** It is also a good idea to compare against a small window rather than a specific value. In this case, accepting any hit between **x=119** to **x=121** and **y=74** and **y=76** might be a good choice.

Using the Gonzo Utilities

I have written a set of **PostScript Gonzo Utilities**. These are totally device independent and offer exceptional typesetting quality and superb drawings in amazingly compact file sizes. Gonzo can be ideal for further post processing of captured .PDF info.

On the other hand, Gonzo is proudly non-WYSIWYG and has a steep learning curve, so it is clearly not for everybody. Many hundreds of **.PSL** Gonzo example files appear in our **GrurGram**, our **PostScript**, and similar **Guru's Lair** libraries.

In a recent client job, a **95K** header and a **35K** page file got replaced with a **2973** byte **Gonzo** file. With considerable slop and bloat remaining if file size is **really** a big time issue.

Typical final code to be distilled might look something like this…

```
(C:\\Documents and Settings\\don\\Desktop\\
        gonzo\\gonzo.ps) run
landscape
/font1 /Helvetica-Bold 9 gonzofont font1
[   [  21.0 563.0 (string100) ]
    [ 187.0 535.0 (string101) ]
       .   .   .   .   .
    [ 187.0 521.0 (string1xx) ]
] {aload pop cl} forall

/font2 /Helvetica 9 gonzofont font2
[  [ 56.0 464.0 (string200) ]
   [ 87.0 464.0 (string201) ]
       .   .   .   .   .
   [ 39.0 464.0 (string2xx) ]
] {aload pop cl} forall

showpage
% EOF
```

An Important Update...

Versions 8.1 and higher of Acrobat Distiller default to preventing general file reads or writes.

The workaround is to run Distiller from the Windows command line using "acrodist -F" . Solutions for other operating systems are found here.

For Additional Assistance

Similar tutorials and additional support materials are found on our **PostScript** and our **GurGram** library pages. As always, **Custom Consulting** is available on a cash and carry or contract basis. As are seminars.

For details, you can email **don@tinaja.com**. Or call **(928) 428-4073**.