

Modifying a Postscript Printer Definition to Produce a Negative Image for Typesetting

Larry Hoyle, University of Kansas
Institute for Public Policy and Business Research

Introduction

The device drivers provided with SAS/GRAPH® are written in such a way that much of their operation can be customized by the user through the changing of default parameters. The GOPTIONS statement provides a simple means for making temporary changes. There are some occasions when a modified driver needs to be used repeatedly. PROC GDEVICE provides the answer for these occasions. It allows the user to create a new device driver entry which is a copy of a SAS Institute supplied driver but which uses modified default settings.

The specific example used in this paper will be a modification of a PostScript driver which produces a negative image on film when output to a typesetter. The example program listed here could be modified for a number of other purposes. Some operating systems, for example require a PostScript file to have a certain prologue to be recognized as PostScript for printing.

The programs listed here have been run with SAS 6.07 under ULTRIX®, AIX®, and VM/CMS®.

The first part of what follows will describe the SAS code necessary to modify a device driver. The remaining part will discuss the PostScript involved in making a negative.

Creating a Graphics Driver Entry

Drivers supplied by SAS Institute are all accessed through the catalog SASHELP.DEVICES. If there are catalogs named GDEVICE_n.DEVICES they are searched first for drivers. To permanently modify an Institute supplied driver, the recommended technique is to first make a copy of it to a catalog such as GDEVICE0.DEVICES and then modify that copy. Listing 1 copies the driver named PS to a new driver named PSNEG.

```
/* Create a new entry in GDEVICE0.DEVICES */
/* The new entry will be a copy of the PS driver from */
/* SASHELP.DEVICES */
libname gdevice0 'system specific';
proc gdevice catalog=gdevice0.devices nots;
  copy ps from=sashelp.devices
    newname=psneg;
end;
run;
```

Listing 1, Creating the New Catalog Entry

```
options source2;
%global maxcols splitcol lnsiz;
/* maxcols is the maximum length of a line */
/* in the PostScript header plus 2 */
%let maxcols=80;
/* splitcol, the maximum length of a string */
/* in the GPROLOG statement */
%let splitcol=99;

/* lnsiz must be splitcol*2+4 or larger */
%let lnsiz=250;

filename modstmt 'system specific';
```

Listing 2, Defining the Global Values

Modifying the New Entry

The next series of listings will show the SAS programs which modify the new device driver entry named PSNEG. Listing 2 shows the global definitions which are common to all of the rest of the SAS programs shown. Listings 2 through 7 might commonly be run all as one job.

GDEVICE is the procedure used to change the parameters of the new driver. In this particular example the GPROLOG parameter of the new PSNEG driver needs to be modified. GPROLOG contains the string which is sent to the graphics device just before the graphics commands. This string should be specified in hexadecimal ASCII. The figure labeled Screen 2 shows the complete GPROLOG generated by the SAS program in Listings 2 through 7.

PROC GDEVICE has both an interactive and a line mode. The GPROLOG string in this example is quite long for interactive entry. Modifications to the hexadecimal string are also quite tedious in interactive mode. Since PostScript is defined in terms of printable characters, the best solution is to use a DATA step to build a command to modify the parameter in line mode. The first step in that process is to create a dataset containing the text of the PostScript prologue. Listing 3 shows the code for that process. Each line of the PostScript prologue is read in as a single string as variable x. The string is then transliterated to ASCII into a string variable a. Finally a carriage return linefeed combination is appended to the end of the string.

```

/* make a dataset of the PostScript prologue. */
/* Each line is a string. */
/* x contains the original line. */
/* a contains the line as ASCII <CRLF> trails. */
data x;
length x a $ &maxcols;
input x $ 1-&maxcols;
alength=length(x)+2;
substr(a,1,length(x)) = put(x,ascii&maxcols.);
substr(a,length(x)+1,2)='D0A'x;

/* ----- The Postscript header follows ----- */
cards;
%IPS-Adobe-1.0
% this section converts everything to a negative image
% first invert the transfer function (black becomes white)
/original-transfer currenttransfer def
{1 exch sub original-transfer} bind settransfer
% then black out the background (the clippath)
gsave
newpath
clippath
1 setgray %note: 1 is now black
fill
grestore
% flip the image
/offset {clippath pathbbox pop exch pop add} def
[-1 0 0 1 offset 0] concat
% this ends the convert to a negative section
%
%
;
/* -----End of the Postscript Header ----- */

```

Listing 3, Reading the PostScript Prologue

Once the PostScript prologue is in dataset form it can be written hexadecimal as part of a MODIFY GPROLOG= statement. As of the writing of this paper there was a "quirk" in SAS/GRAPH which, according to the usage notes for SAS 6.07 under VM/CMS, can cause PROC GDEVICE to produce incorrect values in line mode when modifying GPROLOG, GEPILOG, GSTART, or GEND parameters. The error is produced when a specified hexadecimal value is split across more than six lines. To avoid this quirk, the example will write the MODIFY statement in strings of the maximum possible length. Listing 4 shows the code which will write the appropriate MODIFY statement. This DATA step writes the PostScript prologue one character at a time from the ASCII copy in hexadecimal format. The hexadecimal codes are written as long lines enclosed by a single quote on the left and a single quote followed by an x on the right. The last line is

```

data _null_; /* writes a MODIFY statement */
set x end=last;
retain n 0;
file modstmt linesize=&insiz;
if _n_=1 then do;
put "modify psneg gprolog=";
put "'@";
end;
/* write the hex string */
do i=1 to alength;
n+1;
/* split up long lines */
if mod(n,&splitcol) = 0 then do;
put "x";
put "'@";
end;
y = substr(a,i,1);
put y $hex2.@;
end;
if last then put "x;";

```

Listing 4, Writing the MODIFY statement.

followed by a semicolon.

It is also useful to have a line-by-line listing of the text of the PostScript prologue together with the hexadecimal representation of each line. Listing 5 shows the code which will write that listing to the LOG. Listing 8 shows a section of the LOG as written by the code in

```

/* write documentation to the log */
data _null_;
file log;
retain n 0;
set x;
/* write the line as text */
do i=1 to length(x);
n+1;
y = substr(x,i,1);
put y $2.@;
end;
n+2;
put "<crif> (" n ")";
/* and in hexadecimal */
do i=1 to alength;
y = substr(a,i,1);
put y $hex2.@;
end;
put;

```

Listing 5, Writing lines to the LOG

```

/* Finally, modify the driver */

proc gdevice c=gdevice0.devices nofs;
%include modstmt;
;
modify psneg
  description='Postscript - negative image';
list psneg;
quit;
run;

```

Listing 6, Executing the Modification

```

/* Test the new driver */
libname gdevice0 'system specific';
filename sasneg 'system specific';
goptions device=PSNEG gsfname=sasneg gsfmode=replace;
proc gtestit pic=1;
run;

```

Listing 7, Testing the Driver

Listing 5. The text representation is written with extra spaces so as to line up with the hexadecimal representation. Note that each prologue line ends with a cumulative count of the characters written to the GPROLOG. This count includes carriage returns and line feeds.

The new driver is actually modified when the MODIFY statement which was written to a file is executed as part of a GDEVICE procedure. The description parameter should also be changed to reflect the new driver's function.

Listing 7 shows a sample program which could be used to test the new driver. The driver is selected in a GOPTIONS statement just like any SAS Institute supplied driver. The other graphics options in the example specify that the output is written to a graphics stream file. This file would typically then be sent to the typesetting device, although for testing it could be sent to a laser printer, or a PostScript display device.

```

%!PS-Adobe-1.0 <crLf> (16 )
252150532D41646F62652D312E300D0A
% this section converts everything t
o a negative image <crLf> (75 )
2520202020746869732073656374696F6E20636F6E76657274732065766572797468696E672074
6F2061206E6567617469766520696D6167650D0A
% first invert the transfer function
( black becomes white ) <crLf> (138 )
2520202020666972737420696E7665727420746865207472616E736665722066756E6374696F6E
2028626C61636B206265636F6D6573207768697465290D0A
/ original-transfer currenttransfer def
<crLf> (178 )
2F6F726967696E616C2D7472616E736665722063757272656E747472616E73666572206465660D
0A
{ 1 exch sub original-transfer } bind set
transfer <crLf> (227 )
7B31206578636820737562206F726967696E616C2D7472616E736665727D2062696E6420736574
7472616E736665720D0A
% then black out the background (the
clippath) <crLf> (278 )
25202020207468656E20626C61636B206F757420746865206261636B67726F756E642028746865
20636C697070617468290D0A
gsave <crLf> (285 )
67736176650D0A
newpath <crLf> (294 )
6E6577706174680D0A
clippath <crLf> (304 )
636C6970706174680D0A
1 setgray % note: 1 is now black <crLf> (
340 )
31207365746772617920202020256E6F74653A2031206973206E6F7720626C61636B0D0A
fill <crLf> (346 )
66696C6C0D0A
grestore <crLf> (356 )
67726573746F72650D0A

```

Listing 8, A Section of the SAS Log

PROC GDEVICE, Interactive Mode

Screen 1 shows the DETAIL screen from an interactive PROC GDEVICE session. To get to this point you would first execute the command:

```
PROC GDEVICE catalog=GDEVICE0.DEVICES; RUN;
```

in a DISPLAY MANAGER session. You would next see a list of device entries and enter either an "E" or a "B" to the left of the PSNEG driver to either edit or browse the entry. If the driver is selected in edit mode any of the parameters on the GDEVICE screens can be changed on screen. The "description" field, for example could be easily changed.

Screen 2 shows the GPROLOG screen. It can be selected from the detail screen by typing "GPROLOG" on the command line. These hexadecimal codes can be changed on screen, but errors could be a real risk.

```
GDEVICE: DIRECTORY GDEVICE0.DEVICES (E)
GDEVICE: Detail
Command ---->

Catalog: GDEVICE0.DEVICES                               Entry: PSNEG

Orig Driver: PS                                           Module: SASGDPSL      Model: 1201
Description: Postscript - negative image                  Type: PRINTER
Lrows: 75 Xmax: 8.500 IN Hsize: 7.500 IN Xpixels: 2550
Lcols: 100 Ymax: 11.000 IN Vsize: 10.000 IN Ypixels: 3300
Prows: 100 Horigin: 0.500 IN
Pcols: 75 Vorigin: 0.500 IN
Aspect: 0.000 Rotate:
Driver query: - Queued messages: N
Gprotocol: Paperfeed: 0.000 IN
Gaccess: sasgastd>goutput
Gsfname: Gsfmode: PORT Gsflen: 0
Trantab: Devmap:
Devtype: PRINTER
```

Screen 1, PROC GDEVICE Detail Screen

```
G GDEVICE: Gprolog
G Command ---->
C
C Catalog: GDEVICE0.DEVICES                               Entry: PSNEG
C
O 252150532D41646F62652D312E300D0A2520202020746869732073656374696F6E20636F
D 6E76657274732065766572797468696E6720746F2061206E6567617469766520696D6167
D 650D0A2520202020666972737420696E7665727420746865207472616E73666572206675
L 6E6374696F6E2028626C61636B206265636F6D6573207768697465290D0A2F6F72696769
L 6E616C2D7472616E736665722063757272656E747472616E73666572206465660D0A7B31
P 206578636820737562206F726967696E616C2D7472616E736665727D2062696E64207365
P 747472616E736665720D0A25202020207468656E20626C61636B206F7574207468652062
A 61636B67726F756E64202874686520636C697070617468290D0A67736176650D0A6E6577
D 706174680D0A636C6970706174680D0A31207365746772617920202020256E6F74653A20
G 31206973206E6F7720626C61636B0D0A66696C6C0D0A67726573746F72650D0A25202020
G 20666C69702074686520696D6167650D0A2F6F6666736574207B636C6970706174682070
G 61746862626F7820706F70206578636820706F70206164647D206465660D0A5B2D312030
T 20302031206F666673657420305D20636F6E6361740D0A252020207468697320656E64
D 732074686520636F6E7665727420746F2061206E656761746976652073656374696F6E0D
D 0A250D0A250D0A
```

Screen 2, PROC GDEVICE GPROLOG Screen

The PostScript Prologue

So far we have treated the PostScript code included in the GPROLOG as a "black box". The rest of this paper will discuss how the PostScript prologue actually produces a negative image. A complete description of the PostScript language cannot be presented here. Please consult the *Postscript Language Reference Manual* for a complete description. The *PostScript Language Tutorial and Cookbook* is also a good resource for those learning the language.

PostScript has nice properties for the problem at hand. The modified device driver we created sends some code to the PostScript printer before the actual graphics producing code is sent. This initial code modifies the behavior of the printer throughout the execution of the graphics producing code. The graphics producing code itself is not modified.

This can work because the PostScript interpreter in the output device maintains a "graphics state" within the context of which PostScript statements are interpreted. The

two particular elements of the graphics state our new driver uses are the current transformation matrix and the current transfer function.

The current transformation matrix defines the mapping between the user space (coordinate system) within which the PostScript commands operate and the physical device space used by the output device. Our driver modifies the current transformation matrix to make a mirror image of the original. Once it is modified all future operations take place in the new space.

The current transfer function defines the mapping between the 0 to 1 grayscale space used by PostScript operators and the actual shading used by the output device. Our driver simply defines a new transfer function which subtracts the original input from 1 before feeding it to the original transfer function. This converts white to black and black to white for all subsequent operations.

A Simple Example

Postscript Listing 1 shows a simple Postscript program which produces the graphic in Figure 1. On any line, all text following a percent sign (%) is a comment. Certain comments, though, perform "document structuring" functions. The first five lines are header comments as described in the *PostScript Language Reference Manual*. The first

line and the BoundingBox line in particular are the minimal lines necessary to make the PostScript program "Encapsulated PostScript". Other software (eg. a word processing package) can read these lines and use them to modify the current transformation matrix before and after passing the Encapsulated Postscript on to the output device. In the case of Figure 1, the 7 inch wide original was scaled down to fit in a 3.5" wide box.

PostScript is an extensible language. New commands, or operators, can be defined and then used just like the native operators. The five lines beginning with /arrow define such a new operator (called arrow) which outlines an arrow. The arrow operator is actually used in the last part of the example.

The current graphics state includes a definition for the current point and the current path. The path is a sequence of points, lines, and curves which have been defined by operators such as the rlineto (relative line to) in the arrow operator. Rlineto adds a straight line segment to the path. This segment runs from the current point to the point whose coordinates are offset from the current point by the arguments to the rlineto function. Rlineto leaves the end point of the segment as the new current point.

The first time the arrow function is invoked, for example, the current point is (100,150). The first rlineto in the arrow operator, "300 0 rlineto", defines a segment from (100,150) to (100+300,150). The new point (400,150) becomes the current point.

Note that the arguments to operators always precede them. When the output device interprets the PostScript, it puts each item on a stack. Each operator then adds or removes stack items.

PostScript structured documents are divided into two segments. The first part is a prologue which contains structuring comments and definitions of operators. The second part is a script. The script is where objects are actually placed on the page. The "%EndProlog" statement separates the prologue from the script.

The lines beginning with /Helvetica and ending with "(Howdy World) show" print the text in Figure 1. This text is printed at a location 100 points from the left edge of the page and 160 points up from the bottom of the page. (A

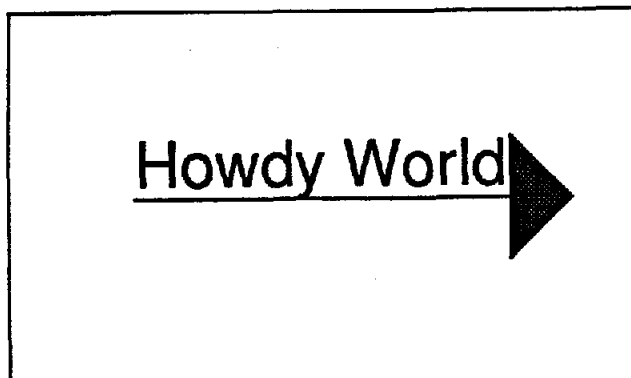


Figure 1, Output from PostScript 1

```
%!PS-Adobe-1.0
%%Title: A Simple Example
%%Creator: Larry Hoyle, University of Kansas
%%BoundingBox: 0 0 500 300
%%EndComments
/arrow {300 0 rlineto
        0 50 rlineto
        50 -50 rlineto
        -50 -50 rlineto
        0 50 rlineto} def
%%EndProlog

%set the font
/Helvetica findfont
50 scalefont
setfont

%select a position
100 160 moveto

%print a string
(Howdy World)show

%draw an arrow
100 150 moveto
arrow
stroke
100 150 moveto
arrow
fill

%%Trailer

%print the page
showpage
```

PostScript 1, A Simple Example

point is 1/72 inch.) The text is 50 point type.

The six lines after the "%draw an arrow" comment actually draw two arrows. The first arrow is put on the page by the stroke operator. The second is put on the page by the fill operator. Stroke puts "ink" on the path, outlining the arrow. Fill puts it inside the path, filling the arrowhead.

The final line in the Postscript Listing 1 example causes the output device to actually put the image on paper or the screen.

Making a negative

The image in Figure 1 is a nice simple one for illustrating the workings of the PostScript prologue used by our device driver. Postscript listing 2 shows the section of code which reverses the shading by modifying the current transfer function. The first line saves a copy of the original transfer function in a new function called "original-transfer." The second line modifies the current transfer function. The new function assumes that the argument for the transfer function is already on the stack (as did the original function), puts 1 on the stack,

```
/original-transfer currenttransfer def
(1 exch sub original-transfer) bind settransfer
```

PostScript 2, Change the Transfer Function

exchanges the two, subtracts the original value from 1, and then applies the original transfer function to the difference. Figure 2 shows the results of including these lines before the code in PostScript listing 1.

The text disappeared in Figure 2 because it is white on a white background. The code in Postscript Listing 3 turns the background black before anything else is placed on the page. The `gsave` command saves a copy of the current graphics state. `Newpath` clears out the current path. `Clippath` makes the

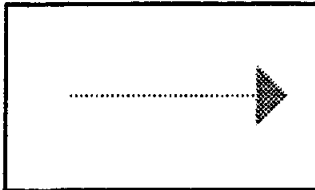


Figure 2, After PS 2

```
gsave
newpath
clippath
1 setgray %note: 1 is now black
fill
grestore
```

PostScript 3, Blacken the Page

current path a box which contains the image area of the whole page. `1 setgray` sets the current color to black. `Fill` fills the current path (the whole page) with the current color (black). `Grestore` restores the graphics state to what it was.

The final step in making a negative is to transform the image into a mirror image of the original. This is necessary so that when the negative is used in the printing process the emulsion is on the right side (against the plate). If the emulsion is on the wrong side the printed page may be slightly blurred. Postscript Listing 4 shows the code which flips the image. The first line defines a function which returns the sum of the width of the current page and the left margin of the current page. `Clippath` followed by `pathbbox` returns a pair of x,y coordinates for the lower left and upper right corners of the page. The `pop exch pop` sequence removes the y coordinates from the stack. The

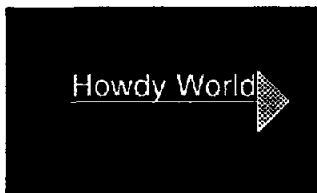


Figure 3, After PS 3

```
/offset {clippath pathbbox pop exch pop add} def
[-1 0 0 1 offset 0] concat
```

PostScript 4, Make a mirror image

`add` operator adds together the two x coordinates. The second line defines a linear transformation of the current space. The x coordinates in the old space are multiplied by the -1. This makes a mirror image with one problem. It is completely off the page to the left. Adding the offset to all x coordinates completes the transformation by moving it to the right by the page width and the left margin. The `concat` operator applies these transformations in one operation.

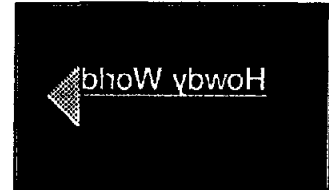


Figure 4, After PS 4

Conclusion

Making a "negative image driver" is just one application of the general technique outlined here. It should be possible, for example, to create four drivers which would output the separated cyan, magenta, yellow, and black components of a color PostScript image for printing a color image. A skewed image of a page, to be used as a reflection, could also be generated. With this technique any effect which relies on an initial change to the graphics state can be incorporated into a new PostScript driver.

Trademark Notice

SAS and SAS/GRAPH is the registered trademark of SAS Institute, Inc. Cary, NC, USA. PostScript and Adobe are trademarks of Adobe Systems Incorporated. ULTRIX, is a trademark of Digital Equipment Corporation. AIX and VM/CMS are trademarks of the IBM corporation.

References

Adobe Systems Incorporated *Postscript Language Reference Manual 2nd. ed.*, Addison-Wesley Publishing Company, 1990.

Adobe Systems Incorporated *Postscript Language Tutorial and Cookbook*, Addison-Wesley Publishing Company, 1985.

Author

Larry Hoyle
Institute for Public Policy
and Business Research
University of Kansas
607 Blake Hall
Lawrence, Kansas 66045-2960
Telephone: (913) 864-3701

