Using EPS Files in PostScript Language Forms

Adobe Developer Support

Technical Note #5144

4 October 1996

Adobe Systems Incorporated

Corporate Headquarters
345 Park Avenue
San Jose, CA 95110
(408) 536-6000 Main Number
(408) 536-9000 Developer Support
Fax: (408) 536-6883

Adobe Systems Eastern Region
24 New England
Executive Park
Burlington, MA 01803
(617) 273-2120
Fax: (617) 273-2336

European Engineering Support Group
Adobe Systems Benelux B.V.
P.O. Box 22750
1100 DG Amsterdam
The Netherlands
+31-20-6511 355
Fax: +31-20-6511 313

Adobe Systems Co., Ltd.
Yebisu Garden Place Tower
4-20-3 Ebisu, Shibuya-ku
Tokyo 150
Japan
+81-3-5423-8169
Fax: +81-3-5423-8204

PN LPS5144

Contents

Using EPS Files in PostScript Language Forms

1 Introduction

In PostScript Level 2, Adobe introduced the **execform** operator to support the caching of graphical objects or "forms" for repeated use. In order to ensure that each execution of a form produces the same output, it must be completely self-contained and conform to several constraints. One of these constraints is that all graphical content must be described within the form's PaintProc. Since the PaintProc is a procedure, it may not contain references to in-line data. This restriction, as well as the implementation size limits for procedures and strings, could present obstacles to placing an encapsulated PostScript (EPS) file in a form.

Using an EPS file in a form is possible, however, provided the user has adequate VM or a writeable storage drive. This paper will explain how to accomplish this, and will provide some emulation code for Level 1 devices.

2 Storing the EPS file in VM

Placing an EPS file into a form's PaintProc is not simply a matter of taking the EPS file and bracketing it with curly braces. This would likely result in PostScript error messages, for a number of reasons. For example, EPS files often contain in-line data for fonts or images, which are inappropriate contents for any procedure. (See the Q&A section of the *ADA News*, volume 3, issue number 1, for an explanation of why in-line data will not work in a procedure body.) Instead, an EPS file can be executed by a PaintProc with the use of the **SubFileDecode** filter. Example 1 below shows how this is done.

2.1 Example 1 Code Walk-through

In example 1, we store the contents of an EPS file in an array of strings. Then, we define a form with a PaintProc that consecutively executes each string in the array. The primary tasks performed by the code are as follows:

1. The readdata procedure, called prior to the inclusion of the EPS file, consumes the EPS file and puts its contents into an array of strings, EPSArray.

We have chosen a string buffer size of 16000 bytes to avoid premature VMerrors as the result of fragmentation of VM.

The initial entry in EPSArray is a 1-element array, containing a counter. The counter is stored in an array so that it can be maintained in global VM, and thereby be unaffected by calls to **save** or **restore** encountered when executing the EPS file. The following *n* entries are strings containing the EPS contents. *n* can be determined by dividing the size of the EPS file by 16000 (the string size) and rounding up. Lastly, an empty string must be present in the array, immediately following the EPS strings, to indicate the end of data.

2. After the EPS file, we place the comment: *% EOD_Marker_4386* in the output file. The **SubFileDecode** filter in the readdata procedure uses this comment as an end-of-data marker. The last four numbers are generated randomly when the file is created. You may choose to use any numbers or characters for this comment; however, it is important that you vary the comment for each job to avoid problems that may occur when nesting executions of **SubFileDecode**.

3. We define the form resource in the document set-up section of the output file.

   The BBox entry of the form has the same value as that offered by the *%%BoundingBox:* comment of the EPS file.

   The PaintProc procedure uses the **SubFileDecode** filter to execute the EPS file. The data acquisition procedure used with **SubFileDecode**, AcquisitionProc, traverses EPSArray, providing the EPS strings to the filter on demand. The execution of the EPS file is contained within the context of the StartEPSF and EPSFCleanUp procedures, which serve to create and restore the proper PostScript environment for the EPS file. These procedures are identical to what would be used to import an EPS file, regardless of its use in a form; this process is described in section H.3.2 of the *PostScript Language Reference Manual, Second Edition*.

4. Within the individual page descriptions, we execute the form by calling the **execform** operator. Three pages, containing the EPS file, are printed in this example.

Note *Prior to executing the code below, your application should determine that there is adequate free VM to store the EPS file. If there is limited free VM, you may need to include the EPS file multiple times within the job, rather than using forms. Alternatively, if the user's output device has writeable storage, you may choose to store the EPS file on disk, as described in section 5 of this document.*

Example 1: *Executing EPS in form's PaintProc*

```
%!PS-Adobe-3.0
%%Title: (Using EPS file in Form in VM)
%%BoundingBox: 0 0 612 792               % UPDATE FOR EACH EPS
%%DocumentProcessColors: Black           % PROMOTE DSC COMMENTS FROM EPS
%%LanguageLevel: 2
%%Pages: 3
%%EndComments

%%BeginProlog
%%BeginResource: procset forms_ops 1.0 0
%%Title: (Forms Operators)
%%Version: 1.0
userdict /forms_ops 10 dict dup begin put
/StartEPSF { % prepare for EPSF inclusion
    userdict begin
      /PreEPS_state save def
      /dict_stack countdictstack def
      /ops_count count 1 sub def
      /showpage {} def
} bind def

/EPSFCleanUp { % clean up after EPSF inclusion
    count ops_count sub {pop} repeat
    countdictstack dict_stack sub {end} repeat
    PreEPS_state restore
    end % userdict
} bind def

/STRING_SIZE 16000 def         % Best value to not fragment printer's VM
/ARRAY_SIZE  40 def            % UPDATE FOR EACH EPS == filesize/16000 + 2
                               % for initial counter and final empty string.

/buffer STRING_SIZE string def
/inputFile currentfile 0 (% EOD_Marker_4386) /SubFileDecode filter def

/readdata { % array readdata --
   1 {                          % put counter on stack
                                % stack: array counter
       2 copy                   % stack:  array counter array counter
       inputFile buffer readstring    % read contents of currentfile into buffer
                                % stack:  array counter array counter string boolean
       4 1 roll                 % put boolean indicating EOF lower on stack
       STRING_SIZE string copy  % copy buffer string into new string
                                % stack:  array counter boolean array counter newstring
       put                      % put string into array
       not {exit} if            % if EOF has been reached, exit loop.
       1 add                    % increment counter
  } loop
  % increment counter and place empty string in next position
  1 add 2 copy () put pop
  currentglobal true setglobal exch

      0 1 array put   % create an array for counter in global VM,
                      % so as not to be affected by save/restore calls in EPS file.
                      % place as first element of string array.
```

```
  setglobal % restore previously set value
} bind def
currentdict readonly pop end
%%EndResource
%%EndProlog

%%BeginSetup
% set MaxFormItem to be equivalent to MaxFormCache
<< /MaxFormItem currentsystemparams /MaxFormCache get >> setuserparams

% make forms procset available
forms_ops begin
userdict begin

% download form resource
%%BeginResource: form TestForm
/TestForm
10 dict begin
   /FormType 1 def
   /EPSArray ARRAY_SIZE array def
   /AcquisitionProc {
           EPSArray dup 0 get dup 0 get % array counter_array counter
           dup 3 1 roll                % array counter counter_array counter
           1 add 0 exch put            % increment counter
           get                         % use old counter as index into array, placing
                                       % next string on operand stack.
   } bind def

   /PaintProc {
       begin
         StartEPSF
         % May want to translate here, prior to executing EPS
           EPSArray 0 get 0 1 put
           //AcquisitionProc 0 () /SubFileDecode filter
           cvx exec
         EPSFCleanUp
       end
   } bind def

   /BBox [ 0 0 612 792] def          % UPDATE FOR EACH EPS
   /Matrix [1 0 0 1 0 0] def
currentdict end def  % TestForm

TestForm /EPSArray get
readdata
%%BeginDocument:  (title.eps)        % UPDATE FOR EACH EPS
  ... put EPS file here ...
%%EndDocument
% EOD_Marker_4386    % Put this comment after your EPS file so
                     % SubFileDecode filter will reach end of data.
%%EndResource
%%EndSetup

%%Page: 1 1
%%BeginPageSetup
/pgsave save def
%%EndPageSetup
TestForm execform
```

```
%%PageTrailer
pgsave restore
showpage

%%Page: 2 2
%%BeginPageSetup
/pgsave save def
%%EndPageSetup
TestForm execform
%%PageTrailer
pgsave restore
showpage

%%Page: 3 3
%%BeginPageSetup
/pgsave save def
%%EndPageSetup
TestForm execform
%%PageTrailer
pgsave restore
showpage
%%Trailer
end % userdict
end % forms_ops
%%EOF
```

Note *The above code does not produce any output as is. To use this code with an EPS file, insert the file where you see the line, "% … put EPS file here …." Make other modifications to the code, as specified by the in-line comments.*

3 Emulating 1-page forms in a Level 1 environment

Technical note #5113, "Emulation of the execform Operator," offers sample code for emulating forms in a Level 1 environment. However, this code cannot be used when the form contains an EPS file. Notice that our technique for including EPS files in forms requires a Level 2 or greater output device, because it requires the use of a filter, **SubFileDecode**, introduced in PostScript Level 2. If your user is printing to a Level 1 device with a writeable storage drive, then you can use the method described in section 5 of this document to emulate forms. Otherwise, depending on how you are using the form, you may be able to use the **copypage** operator to achieve needed results in a Level 1 environment. If you are using PostScript forms to print an EPS file several times on one page, then the only option in Level 1 is to send the EPS file to the printer multiple times. However, if you are using the form once per page, for example, for a forms fill-out program, then **copypage** can be used as a Level 1 alternative.

Unfortunately, **copypage** has the severe disadvantage that it prevents page independence. For this reason, DSC-conforming files that use **copypage** must always use the "%%PageOrder: Special" header comment to alert consumers that the file is page dependent. Programs that reorder pages or perform imposition should reject page-dependent files. If at all possible, your

application should detect when the user is printing to a Level 2 or greater printer, or provide a Level 2-only option to the user, so that page-independent Level 2 code can be used instead.

Example 2 below uses **copypage** on Level 1 devices and PostScript forms on all other devices. **copypage** should only be used in Level 1 environments; since the release of PostScript Level 2, use of **copypage** is *strongly discouraged*. Appendix I of the *PostScript Language Reference Manual, Second Edition* states,

> "The **copypage** should not be used to simulate forms functionality; use the **execform** operator".

PostScript forms offer better performance and reliability than the **copypage** operator and should always be used when available.

3.1 Example 2 Code Walk-through

Example 2 demonstrates how a typical form fill-out application may work. The code determines if it is executing in either a Level 1 or a Level 2 or greater environment. Various calls and definitions are conditionally made depending on the environment encountered.

1. Define form or execute EPS file.

   In the case of a Level 2 or greater device, the code is similar to Example 1 – a form dictionary is defined, and the readdata procedure stores the EPS file in an array of strings to be used by this form. On a Level 1 device, the EPS code is executed when encountered in the file. StartDoc and EndDoc procedure calls surround the execution of the EPS file in both cases.

2. Add data to fields in form.

   Data is added to the form with a call to the add_var_data procedure. This procedure takes four strings as its operands and prints those strings in the appropriate places on the form. For the Level 1 case, add_var_data must clear the data field area by painting white boxes in those fields; this is necessary for the second and subsequent pages, since the data from the previous page will need to be erased. This step is not necessary when the PostScript forms method is used.

3. Emit page.

   To emit a page on a Level 2 device, **execform** is called, followed by a call to **showpage**. On a level 1 device, **copypage** is called, except for the final page, in which case **showpage** is called

Note *When printing to a Level 1 device, the contents of the included EPS file must be Level 1-compatible.*

Example 2: *Form emulation using copypage*

```
%!PS-Adobe-3.0
%%Title: (Using EPS file in Form in VM)
%%BoundingBox: 0 0 612 792
%%DocumentProcessColors: Black
%%Pages: 3
%%PageOrder: Special
%%EndComments

%%BeginProlog
%%BeginResource: procset helper_ops 1.0 0
%%Title: (Helper Operators)
%%Version: 1.0
userdict /helper_ops 10 dict dup begin put

/L1? {
    /languagelevel where {
       pop languagelevel 2 lt
    }{
       true
    } ifelse
} bind def

/StartDoc {
   userdict begin
   /PreForm save def
   /showpage {} def
} bind def

/EndDoc {
   PreForm restore
   end % userdict
} bind def

/rfill { % llx lly w h rfill --
   gsave newpath
   4 -2 roll moveto % use x, y coordinates
   2 copy 0.0 lt exch 0.0 lt xor { % check for one arg only being neg
       dup 0.0 exch rlineto % do height first
       exch 0.0 rlineto
       neg 0.0 exch rlineto
   }{
       exch dup 0.0 rlineto % do width first
       exch 0.0 exch rlineto
       neg 0.0 rlineto
   } ifelse
   closepath
   fill grestore
} bind def

/add_var_data { % procedure to add variable data to fill in form
    /Times-Roman findfont 12 scalefont setfont
    L1? { % if Level 1, need to draw white boxes where data will be placed
```

```
        gsave
        1 setgray
        87 360 350 20 rfill
        190 430 350 20 rfill
        190 535 350 20 rfill
        190 600 350 20 rfill
        grestore
    } if
    97 370 moveto show
    200 440 moveto show
    200 545 moveto show
    200 610 moveto show
} bind def

/emitpage {
   L1? {
      add_var_data copypage
   }{
     TestForm execform add_var_data showpage
   } ifelse
} bind def

/emitlastpage {
   L1? {
      add_var_data showpage
   }{
      emitpage
   } ifelse
} bind def

currentdict readonly pop end
%%EndResource

%%BeginResource: procset forms_procs 1.0 0
%%Title: (Forms Procs)
%%Version: 1.0
userdict /forms_procs 10 dict dup begin put

/STRING_SIZE 16000 def
/ARRAY_SIZE  5 def            % UPDATE DEPENDING ON SIZE OF INCLUDED DOCUMENT
                              % (leave room for initial counter and final empty string)

/buffer STRING_SIZE string def

/readdata {

  L1? {  % readdata --
      StartDoc
  }{     % array readdata --
      1 {                         % put counter on stack
                                  % stack: array counter
           2 copy                 % stack:  array counter array counter
           inputFile buffer readstring    % read contents of currentfile into buffer
                                  % stack:  array counter array counter string boolean
           4 1 roll               % put boolean indicating EOF lower on stack
           STRING_SIZE string copy  % copy buffer string into new string
                                  % stack:  array counter boolean array counter newstring
           put                    % put string into array
```

```
           not {exit} if            % if EOF has been reached, exit loop.
           1 add                    % increment counter
       } loop
       % increment counter and place empty string in next position
       1 add 2 copy () put pop
       currentglobal true setglobal exch

           0 1 array put   % create an array for counter in global VM,
                           % so as not to be affected by save/restore calls in EPS file.
                           % place as first element of string array.
       setglobal % restore previously set value
  } ifelse
} bind def
currentdict readonly pop end
%%EndResource

%%EndProlog
%%BeginSetup
helper_ops begin
forms_procs begin
userdict begin

L1? not {
   % set MaxFormItem to be equivalent to MaxFormCache
   1 dict begin
       /MaxFormItem
          currentsystemparams /MaxFormCache get
       def
   currentdict end
   setuserparams
}if

% download form
%%BeginResource: form TestForm
L1? not { % for > L1 case, define a form resource
/TestForm
10 dict begin
   /FormType 1 def
   /EPSArray ARRAY_SIZE array def
   /AcquisitionProc {
           EPSArray dup 0 get dup 0 get % array counter_array counter
           dup 3 1 roll                % array counter counter_array counter
           1 add 0 exch put            % increment counter
           get                         % use old counter as index into array, placing
                                       % next string on operand stack.
   } bind def

   /PaintProc {
       begin
         StartDoc
           EPSArray 0 get 0 1 put
           /AcquisitionProc load 0 () /SubFileDecode filter
           cvx exec
       end % form dict
   } bind def
   /BBox [ 29 222 583 759] def       % UPDATE FOR INCLUDED DOCUMENT
   /Matrix [1 0 0 1 0 0] def
currentdict end def  % TestForm
```

```
/inputFile currentfile 0 (% EOD_Marker_5882) /SubFileDecode filter def
TestForm /EPSArray get  } if

readdata
%%BeginDocument: landscape_form      % UPDATE TO NAME OF INCLUDED DOCUMENT
%!PS-Adobe-3.0 EPSF-3.0
%%Title: (sample landscape form)
%%CreationDate: (9/4/96) (4:48 PM)
%%BoundingBox: 29 222 583 759
%%DocumentProcessColors: Black
%%DocumentFonts: Helvetica-Bold
%%EndComments
%%BeginProlog
%%BeginResource: procset simple_graphics 1.0 0
%%Title: (Abbreviated Operators)
%%Version: 1.0
20 dict begin
  /m /moveto load def
  /l /lineto load def
  /L /lineto load def
  /S /stroke load def
  /s /stroke load def
  /g /setgray load def
  /f /fill load def
  /F /fill load def
  /w /setlinewidth load def
currentdict end /my_dict exch def
%%EndResource
%%EndProlog
%%BeginSetup
my_dict begin
%%EndSetup
%%Page: 1 1
%%BeginPageSetup
/pgsave save def
%%EndPageSetup

0 g
110.3759 735.6955 m 103.0572 728.2942 l 108.4972 728.3248 l
101.3305 722.3643 l 110.4505 722.4156 l 119.4104 722.4659 l
111.7768 728.4232 l 117.7768 728.4569 l 110.3759 735.6955 l
f
109.3986 723.2999 m 108.3653 719.9349 l 110.8125 719.1834 l
111.9398 722.8543 l
F

0.5 g
375 661 m 375 675 L 34 675 L 34 661 L 372 661 L
f
0 g

2 w
375 660 m 375 754 L 34 754 L 34 660 L 375 660 L
s
204.5 707 m
S
```

```
383 739 moveto
1 w
/Helvetica-Bold findfont 12 scalefont setfont
(ORDER\r) show
383 710 moveto
(REQUEST\r) show
383 681 moveto
(FORM) show

0.5 w
385 729 m 568 729 l S
385 699 m 568 699 l S
385 669 m 568 669 l S

2 w
578 227 m 578 650 L 34 650 L 34 227 L 578 227 L s
306 441 m S

57 601 m
(NAME:) show

0.5 w
565 595 m 565 638 L 50 638 L 50 595 L 565 595 L s
57 484 m (ADDRESS:) show

565 478 m 565 584 L 50 584 L 50 478 L 565 478 L s
57 429 m (PHONE:) show

565 423 m 565 466 L 50 466 L 50 423 L 565 423 L s
57 393 m (SERVICES REQUESTED:) show

565 239 m 565 411 L 50 411 L 50 239 L 565 239 L s
49.5 694.5 m
0.5 g
/Times-Bold findfont 30 scalefont setfont
(LEO S LANDSCAPING) show
46 698 m
0 g
(LEO S LANDSCAPING) show

86 663 m
1 g
/Helvetica-Bold findfont 12 scalefont setfont
(PROFESSIONAL LAWN & GARDEN CARE) show

%%PageTrailer
pgsave restore showpage
%%Trailer
end % my_dict
%%EOF
%%EndDocument
EndDoc                % put this call after included EPS file. Followed by
% EOD_Marker_5882     % The % EOD_Marker_#### comment, which is needed so that
                      % SubFileDecode filter will reach end of data.

%%EndResource
%%EndSetup
```

```
%%Page: 1 1
%%BeginPageSetup
/pgsave save def
%%EndPageSetup
(Joe Smith)
(1281 Market Street, San Jose, CA 95053)
(408-225-1818)
(Trim two pine trees in NE corner of lot.)
emitpage
%%PageTrailer
pgsave restore

%%Page: 2 2
%%BeginPageSetup
/pgsave save def
%%EndPageSetup
(Patty Davis)
(1441 East Willow Circle, Tucson, AZ 44388)
(210-555-1212)
(Plant 5 Azaleas under kitchen bay window.)
emitpage
%%PageTrailer
pgsave restore

%%Page: 3 3
%%BeginPageSetup
/pgsave save def
%%EndPageSetup
(Beatrice Miller)
(410 Park Avenue, New York, NY 12023)
(212-584-6789)
(Remove diseased palm tree. Replace with 10 gallon spruce.)
emitlastpage
%%PageTrailer
pgsave restore

%%Trailer
end % userdict
end % forms_procs
end % helper_procs
%%EOF
```

Figure 1 *Output from first page of example 2*

4 Performance Advantage of PostScript Forms

In addition to the major advantage of offering page-independence, and the flexibility of printing multiple forms on one page, PostScript forms also offer a performance advantage over using the **copypage** operator. If a form is not cached due to limited memory on the printer, then forms output can be slower than **copypage** use. However, when form caching occurs, forms may be 35% to 90% faster than using **copypage**. Table 1 below shows the print times for several EPS files, comparing the two printing methods. The EPS files range from simple fill-out form templates, such as IRS tax forms, to complex Adobe Illustrator® graphics containing multiple objects and gradients.

The EPS files tested are as follows:

Type poster, Power drill, and Henry's trip – Adobe Illustrator sample files from the Illustrator 6.0 product CD for Macintosh.

Hands, Truck and Window – Adobe Photoshop™ sample files from the Photoshop 3.0 product CD for Macintosh. Hands is a 468 x 414 pixel gray-scale image; Truck is a 499 x 344 pixel RGB image; and Window is a 278 x 414 pixel RGB image.

Forms 1 through 4 – a selection of form templates. Form 1 is a half-page IRS W-2 form template; forms 2 and 3 are pages 1 and 2 of an IRS W-5 form template; and form 4 is a company's 401K investment form template.

The timing results below were made by placing calls to the **realtime** operator before and after the execution of a 10-page document, where the EPS file was printed once per page. The forms were tested on two printers, a PostScript version 2013.112 and a PostScript version 2014.106 printer. Both times are listed next to the filename.

Table 1 *Timing results comparing copypage and forms usage*

filename	*time using copypage (ms)*	*time using forms (ms)*	*% faster with forms*
Hands	**110,608**	**40,704**	**63%**
	64,384	**19,024**	**70%**
Henry's trip	**119,760**	**42,984**	**64%**
	69,736	**20,440**	**71%**
Truck	**245,872**	**160,712**	**35%**
	114,632	**30,752**	**73%**
Power drill	**113,928**	**59,416**	**48%**
	111,664	**30,816**	**72%**
Type poster	**115,648**	**46,776**	**60%**
	60,360	**15,512**	**74%**
Window	**185,992**	**107,048**	**42%**
	71,528	**26,520**	**63%**
Form 1	**146,720**	**93,192**	**36%**
	75,504	**35,200**	**53%**
Form 2	**195,680**	**119,936**	**39%**
	88,184	**49,336**	**44%**
Form 3	**173,336**	**100,216**	**42%**
	77,920	**37,528**	**52%**
Form 4	**82,328**	**10,784**	**87%**
	47,208	**4,360**	**91%**

5 Storing an EPS on a writeable drive, for use in a Form

Storing an EPS file in PostScript VM may be a reasonable solution for a job that only uses one form. However, some EPS files may be too large to fit into a printer's available VM. Furthermore, there may be situations where a user wants to use several complex EPS files on multiple pages. Or, a user may wish to use EPS files in forms that need to be cached across several jobs. Fill-

ing up VM with EPS files is probably not the best solution in this case. A better solution would be to store multiple complex forms on the device's writeable storage drive. If the user does not have adequate VM or a writeable drive to store the EPS file, your application should fall back to sending the EPS file down to the printer each time it is needed.

Rather than storing the EPS file in an array in VM, your application may temporarily store the EPS file on the printer's disk, use it in the form, and then delete the file at the end of the print job. Example 3 below demonstrates how to do this. If the user wants to download EPS files to be used in multiple jobs, you may provide them with a utility to do so.

Example 3 is designed to work only on Level 2 devices with writeable storage drives. The code could be modified, however, to work on a Level 1 printer with a writeable drive. To modify Example 3 to work in a Level 1 environment, first substitute calls to **execform** with the emulation code provided in technical note #5113, "Emulation of the execform operator." Secondly, remove any reference to the **SubFileDecode** operator. This can be handled one of two ways: 1) emulate the **SubFileDecode** filter using code that reads from currentfile one line at a time, using **readline**, and checks for the end-of-data comment; or 2) modify the code so it works as three print jobs. The first job downloads the file to the printer's storage drive. The next job uses the file in a form, and the last job deletes the EPS file from the drive. Following the above guidelines, Example 3 can be modified to be a preferable Level 1 solution, rather than using **copypage**.

5.1 Steps for Writing an EPS file to disk for use in a form

Here are the tasks your application will need to perform:

1. Import the EPS file, using normal means.

2. Store the bounding box value to use later in the form definition.

3. Determine whether the printer is a level 2 printer with a writeable storage drive.

   You can do this by sending query code to the printer. Sample code for querying for a writeable storage drive can be found in the PostScript Technologies column of *ADA News* newsletter, volume 5, issue 7. If you do not have bi-directional communication with the printer, you can ask the user to indicate if the destination printer is a Level 2 printer with a writeable drive. If the printer's PPD file has keyword *FileSystem with corresponding value False, or *LanguageLevel keyword with corresponding value '1', then the printer is either a Level 1 device or does not support file systems. In either case, the code in Example 3 will not work on such a device.

4. When writing out your PostScript language output, use code similar to Example 3 below to:

   a. write the EPS file to the printer's hard drive,
   b. define a form which executes the EPS file,
   c. execute the form throughout the job, by calling **execform,** and
   d. remove the EPS file from the printer's hard drive.

Example 3: *Using EPS file on a writeable drive as the contents of a form*

```
%!PS-Adobe-3.0
%%BoundingBox: 35 35 427 457       % This will vary w/each job
%%DocumentProcessColors: Cyan      % This will vary w/each job
%%LanguageLevel: 2
%%EndComments

% Example 3 takes a simple EPS file, which draws a Cyan square, and prints it
% 4-up on a single page, using forms.

%%BeginProlog
%%BeginResource: procset formtodisk_ops 1.0 0
%%Title: (Operators for Writing Form to Disk)
%%Version: 1.0
userdict /formtodisk_ops 10 dict dup begin put

/StartEPSF { % prepare for EPSF inclusion
    /PreEPS_state save def
    /dict_stack countdictstack def
    /ops_count count 1 sub def
    userdict begin
    /showpage {} def
} bind def

/EPSFCleanUp { % clean up after EPSF inclusion
    count ops_count {pop} repeat
    countdictstack dict_stack sub {end} repeat
    PreEPS_state restore
} bind def

/InputFile currentfile 0 (% End_Of_Data)
/SubFileDecode filter def
/Inbuf 4000 string def  % create string to use as buffer.

/WriteInfo { % Writes contents of currentfile to disk until it reaches end of data
             % marker, which we have specified to be End_Of_data.

     {  WriteTarget
       InputFile Inbuf readstring   % read contents of InputFile into Inbuf
        3 1 roll                    % put boolean indicating EOF at bottom of stack
        writestring                 % write contents of Inbuf into file on disk
        not {exit} if               % if EOF has been reached, exit loop.
     } loop
     WriteTarget closefile
} bind def
currentdict readonly pop end
%%EndResource
```

```
%%EndProlog
%%BeginSetup
formtodisk_ops begin
userdict begin

% create file named MyEPS in Files dir with write privileges
/WriteTarget (/Files/MyEPS) (w) file def
%%EndSetup
%%Page: 1 1
%%BeginPageSetup
/pg_one save def
%%EndPageSetup

WriteInfo
%%BeginDocument: cyansqr.eps      % This will vary w/each job
%!PS-Adobe-3.0 EPSF-3.0
%%Title: (cyansqr.eps)
%%BoundingBox: 10 10 82 82
%%DocumentProcessColors: Cyan
%%EndComments
%%Page: 1 1
%%BeginPageSetup
/pgsave save def
%%EndPageSetup

1 0 0 0 setcmykcolor
10 10 moveto 72 0 rlineto 0 72 rlineto
-72 0 rlineto closepath fill
%%PageTrailer
pgsave restore showpage
%%Trailer
%%EOF
%%EndDocument
% End_Of_Data  % put this marker after your EPS file so
               % SubFileDecode filter will reach end of data.

% Define form which executes EPS file.
10 dict begin
   /FormType 1 def
   /PaintProc
   {
     pop % remove form dictionary from stack
     StartEPSF
       -10 -10 translate % move lowerleft corner of EPS to origin.
       (/Files/MyEPS) run
     EPSFCleanUp
   } def
   /BBox [0 0 72 72] def   % Based on the EPS bounding box.
                           % Should be [0 0 urx-llx ury-lly ]
   /Matrix [1 0 0 1 0 0] def
currentdict end
/EPSForm exch def

gsave
35 35 translate
   EPSForm execform
0 350 translate
   EPSForm execform
```

```
320 0 translate
   EPSForm execform
grestore
355 35 translate
   EPSForm execform

%%PageTrailer
pg_one restore showpage
%%Trailer
(/Files/MyEPS) deletefile
end % userdict
end % formtodisk_ops
%%EOF
```

5.2 Fine Tuning Example 3

As you are fine tuning your code, keep the following in mind.

1. Cleaning up after an error occurs

Using the code in Example 3, if your print job fails after partial execution, a file or partial file may remain on the printer's hard drive. Your code should detect errors and delete any remnant files on the disk, in the event of a PostScript error.

2. Proper file naming

Your application should not use the same filename for each print job, because multiple users may be printing to the same printer simultaneously, using your application. This would cause one or more users to experience problems, such as file access errors. You may want to choose some part of the user's system name or id as a portion of the filename to prevent such an occurrence.

If you offer your users a form-downloading utility, then allow them to name the files, so they will recognize the filename if they choose to delete it later.

3. Checking for adequate storage space

Even though a printer may have a writeable drive attached, this doesn't ensure that the drive has adequate storage for the user's EPS file(s). You should handle the event of insufficient storage space by either querying for a large amount of free space in your query code, or doing a query prior to sending the job to check for estimated amount of storage that will be required by the EPS file(s). A call to **devstatus** with the devicename on the stack will return the amount of free storage space available on that device.

6 Setting the Form Cache

When PostScript forms were first introduced, their primary perceived use was for forms fill-out applications, to be used for tax forms, medical forms, and the like. These types of forms probably do not need more than 100,000 bytes of cache space, the default value of **MaxFormCache** for many devices. The trend more recently is to use Postscript forms for the personalization of more complex graphics, such as contained in Adobe Illustrator or Photoshop files. These more complex graphics require more storage than the default cache permits. Table 3 shows the amount of cache needed by some complex Adobe Illustrator and Photoshop files, as compared with more traditional form templates. Cache size consumed by a form can vary from device to device. To demonstrate this we offer the cache size for two devices: a PostScript version 2013.112 device and a version 2014.106 device. The files used for the cache comparisons are the same files used in the timing tests, described in section 4.

Table 2 *Cache sizes for various types of forms*

filename	*cache consumed on 2013.112 printer (bytes)*	*cache consumed on 2014.106 printer (bytes)*
Henry's trip	**91,196**	**92,264**
Power drill	**240,080**	**241,148**
Type poster	**268,752**	**268,796**
Hands	**199,120**	**199,164**
Truck	**182,736**	**182,780**
Window	**125,392**	**125,436**
Form 1	**21,208**	**9,724**
Form 2	**91,656**	**78,956**
Form 3	**24,576**	**26,108**
Form 4	**13,776**	**13,820**

If your users want to use complex graphics in their forms, you may provide them with a utility to increase **MaxFormCache**. This value cannot be increased within your page description because it is a system parameter, which will persist beyond the scope of the print job. Your utility may send the following code, on the user's request, to increase **MaxFormCache**:

```
<< /Password 0 /MaxFormCache 250000>> setsystemparams
```

The above example increases the form cache to a maximum of 250,000 bytes. The password value may be non-zero if specified by the printer manufacturer or modified by the system administrator.

All of the examples in this paper also contain the following code, or its equivalent, to increase **MaxFormItem** to the same value as **MaxFormCache**:

```
<< /MaxFormItem currentsystemparams /MaxFormCache get >>
setuserparams
```

7 Debugging Your Test Files

7.1 Emulating the execform operator

Debugging your forms code can be frustrating. You probably won't receive intelligent error messages back from the PostScript interpreter, if any, if your job fails somewhere in the form's PaintProc. For example, if your EPS file contains an undefined value, you would get the following message:

```
%%[ Error: undefined; OffendingCommand: Testform ]%%
```

indicating that an undefined error occurred some time during the executing of Testform by **execform**.

You may wish to substitute a call to **execform** with a procedure call, when an error is encountered. For example, you can replace the line:

```
TestForm execform
```

in Example 1, with:

```
gsave
   TestForm dup dup dup
   /Matrix get concat
   /BBox get rectclip
   newpath
   readonly
   /PaintProc get exec
grestore
```

The above code performs steps similar to the **execform** operator, as described in section 4.7.1 of the *PostScript Language Reference Manual, Second Edition*. Although no caching will occur using this code, it will be helpful for debugging purposes as you will receive more meaningful error messages. Going back to our previous example of having an undefined name in our file; resending the file using the code above yields a more meaningful error message:

```
%%[ Error: undefined; OffendingCommand: undefined_name ]%%
```

7.2 Testing whether a form is cached

Section 6.1 of technical note #5113, “Emulation of the execform Operator,” offers sample code for determining the amount of cache consumed by a form. You may use this code to determine if a form is cached. A return value of ‘0’ indicates the form was not cached.

An alternative test, if you have bi-directional communication with the printer, is to put a print statement somewhere in your PaintProc. For example, you could add the line: “(I’m in the PaintProc) print flush” to your PaintProc. You will get this message back from the printer every time the procedure is interpreted. When the cached form is used, you will not receive this message.

8 Summary

Using the code in this paper, you may offer your users the performance benefit of PostScript forms, along with the flexibility of imported EPS files. For forms fill-out applications, **copypage** may be used to emulate forms on Level 1 devices. However, PostScript forms should be used on Level 2 and greater devices to allow page-independence, as well as the flexibility of printing a form multiple times on a single page.

Index