Emulation of the execform Operator

Adobe Developer Support

Technical Note #5113

31 March 1992

Adobe Systems Incorporated
Adobe Developer Technologies
345 Park Avenue
San Jose, CA 95110
http://partners.adobe.com/
Copyright © 1991–1992 by Adobe Systems Incorporated. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the publisher. Any software referred to herein is furnished under license and may only be used or copied in accordance with the terms of such license.

PostScript, the PostScript logo, Adobe, and the Adobe logo are trademarks of Adobe Systems Incorporated which may be registered in certain jurisdictions. Other brand or product names are the trademarks or registered trademarks of their respective holders.

This publication and the information herein is furnished AS IS, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes and noninfringement of third party rights.
## Contents

Emulation of the execform Operator

1 Introduction 5

2 Example Form Definition 6

3 Emulating the execform Operator 8
   Overlaying Data Onto a Form 10
   Working With a Large PaintProc 11

4 Applications Using Forms 13
   Form Stamping Applications 13
   Forms Management Software 15

5 Form Coordinate Space 18

6 Managing the Cache Size 18
   Determining a Form’s Cache Size 19

7 Converting Existing Files to Forms 20

8 Limitations 20

Appendix: Changes Since Earlier Versions 21

Index 23
Emulation of the execform Operator

1 Introduction

In PostScript™ Level 2, a form is a self-contained description of any combination of graphics, text, and images. During its first execution, the form can be cached in a manner similar to the caching of PostScript language fonts. This allows for the fastest possible rasterization of subsequent invocations of the form.

This document discusses the general use of PostScript Level 2 forms, the Level 2 execform operator, and the emulation of execform to allow forms usage to be consistent on both Level 1 and Level 2 devices.


Topics in this document include

- Example form definition
- Emulating execform on a Level 1 device
- Methods of form execution
- Caching forms across jobs
- The printer’s job server loop
- The resource mechanism
- Managing the cache size

Common uses of forms include both full page forms, such as an insurance form or a job application form, and partial page forms, which may be “stamped” several to a page or perhaps one per page, similar to having a logo on each page of a company document or slide presentation. Both techniques are discussed in this document.
2 Example Form Definition

As described in the PostScript Language Reference Manual, Second Edition, a form definition is a PostScript language dictionary containing a few required and possibly a few optional definitions. The required dictionary entries are:

- **FormType** must be 1.
- **BBox** is the bounding box of the form definition.
- **Matrix** is the transformation matrix that maps from the form’s coordinate space to the user space.
- **PaintProc** is a PostScript language procedure that describes the look of the form.

The dictionary entry we are most concerned with is the **PaintProc**. Note that a **PaintProc** can describe arbitrary PostScript language objects including text, graphics, and images. The only rules a **PaintProc** must follow are that the form must be self-contained (see section 4.7 in the PostScript Language Reference Manual, Second Edition for details), and the **PaintProc** must follow the guidelines in Appendix I of the same book.

Additionally, when designing an application program that allows you to save a form as a portable entity (similar to an EPS file), steps must be taken to assure portability and compatibility. See Appendices D, G, and I of the PostScript Language Reference Manual, Second Edition. Additionally, understanding and adhering to the concepts of EPS Files as discussed in Appendix H will also help assure that forms will be portable.

The following PostScript language code fragment defines a form whose **PaintProc** describes a clothing size tag as shown in Figure 1. (The drawing operators have been excluded for brevity.)

```
4 dict dup /totFm exch def
begin % Tots Clothing Tag dictionary
  /FormType 1 def
  /BBox [0 0 250 150] def
  /Matrix [1 0 0 1 0 0] def
  /PaintProc {
    pop
    ... the PostScript language code that draws the form ...
  } bind def
end
```
The `execform` operator requires the form dictionary as a parameter and pushes this dictionary onto the operand stack before executing the `PaintProc`. If the `PaintProc` requires access to any of the definitions in the form dictionary then the `PaintProc` can execute a `begin` on the dictionary parameter, use the data, and then execute an `end` before terminating the `PaintProc`. If, as in the case above, data from the dictionary is not needed, the `PaintProc` can `pop` the dictionary off the operand stack.

A `PaintProc` can be called more than once during a given job or series of jobs. Therefore, allocation of memory in the `PaintProc` is not prudent. Avoid the `def`, `array`, `string` and `dict` creation operators as well as the `[]`, `()`, `< >`, `<< >>`, and `<~ ~>` construction operators. When possible, these items should be allocated once in the form dictionary and then referenced in the `PaintProc` any number of times without re-allocating memory. For example:

```
6 dict dup /formName exch def
begin % Tots Clothing Tag dictionary
 /FormType 1 def
 /BBox [0 0 250 150] def
 /Matrix [1 0 0 1 0 0] def
 /sStr 25 string def % scratch string
 /sArray 5 array def % scratch array
 /PaintProc {
   pop
   ... some code ...
   sStr 0 89 put % use the memory allocated in the dict def
   sStr 1 69 put
   sStr 2 83 put
   sStr show
   ... more code ...
   sStr 0 78 put % re-use the same memory
   sStr 1 79 put
   sStr show
   ... the rest of the code ...
 } bind def
end
```

Figure 1  Output of the totFm form
3 Emulating the execform Operator

The following line of code will paint totFm on a Level 2 device.

\[
\text{totFm execform}
\]

Because \textbf{execform} is a Level 2 feature, this code will produce an error on a Level 1 device unless a Level 1 emulation of the \textbf{execform} operator is provided. Along with the emulation, we need to provide a Level 2 veneer so that the syntax to execute the form code is the same regardless of the language level of the target printer.

The following code fragment defines the Level 2 veneer or the Level 1 emulation procedure \texttt{EF} according to the PostScript language level of the device on which the code is being executed. If the device language level is less than 2

- the graphics state is saved with the PostScript language operator \texttt{gsave},
- the form dictionary is pushed onto the dictionary stack with the operator \texttt{begin},
- the form matrix is concatenated with the CTM,
- a clip path is set up according to the form \texttt{BBox},
- and finally, the \texttt{PaintProc} is executed.

Otherwise, if the device language level is greater than 1, the \texttt{execform} operator is loaded into \texttt{EF}. For additional information on conditionally defining emulations, see Appendix D of the \textit{PostScript Language Reference Manual, Second Edition}.

\begin{verbatim}
%!PS-Adobe-3.0
%!Title: Example of form emulation
%!EndComments
%!BeginProlog
%!BeginResource: procset Adobe_Forms 1.0 0
{/Matrix [] def % to prevent bind problems...}
{/PaintProc {} def % ... initialize vars}
{/BBox [] def}
/languagelevel where
{pop languagelevel} {1} ifelse
2 lt { % if Level 1
/EF { gsave newpath dup begin Matrix concat
  BBox dup dup 0 get exch 1 get moveto % clip to BBox
dup dup 0 get exch 3 get lineto
dup dup 2 get exch 3 get lineto
dup 2 get exch 1 get lineto
closepath
clip newpath
PaintProc end grestore} bind def }
{/ % else}
\end{verbatim}
/EF /execform load def
} ifelse
%%EndResource
... other procset definitions and the rest of the document...

To invoke the form totFm that was defined previously, execute

10 10 translate totFm EF
0 170 translate totFm EF
0 170 translate totFm EF
0 170 translate totFm EF
310 0 translate totFm EF
0 -170 translate totFm EF
0 -170 translate totFm EF
0 -170 translate totFm EF

As shown in Figure 2, the above code fragment paints or “stamps” eight instances of the form defined by the totFm dictionary, onto the page. The first instance of execform (called using EF) validates the form dictionary and makes it read-only. It then calls the PaintProc to paint the form. The subsequent executions of execform paint the form by either invoking the form’s PaintProc or by substituting previously cached output.

Figure 2 totFm “stamped” on a page
3.1 Overlaying Data Onto a Form

In the previous example, all of the clothing tag forms are for size 2. In this case, the number 2 is part of the form definition. A more practical application might be to keep the size number data separate from the form definition and overlay the form with whatever size number is needed. This way you can supply any size number for each instance of the form, thus allowing several different sizes per page.

The data to be overlaid onto a form should be placed relative to the origin of the form and will be translated around the page along with the translation of the form instance.

The following code example places a form called \texttt{fm} with a \texttt{translate}, executes the form using the \texttt{EF} procedure defined in this section, and then places the overlaid data with the procedure \texttt{PD}.

The procedure \texttt{PD} takes a size number value as a parameter, places the value into the string \texttt{str} with the \texttt{put} operator, moves to the proper location relative to the form origin, and then images the string with the \texttt{show} operator.

\begin{verbatim}
/tx /translate load def % save on comm & storage overhead
/str 1 string def % dummy str must be as long as largest data
% string and defid in the form dict

/PD { % data PD --
  str exch 0 exch put % put data into str
  150 25 moveto % correct relative location
  str show
} bind def

... 10 10 tx fm EF
50 PD % 50 is the ASCII code for character “2”
0 170 tx fm EF
51 PD
0 170 tx fm EF
52 PD
0 170 tx fm EF
53 PD
310 0 tx fm EF
54 PD
0 -170 tx fm EF
55 PD
0 -170 tx fm EF
56 PD
0 -170 tx fm EF
57 PD
...

The result is the form \texttt{fm} stamped about the page, each instance overlaid with different data values (2 through 9).
\end{verbatim}
3.2 Working With a Large PaintProc

The form’s **PaintProc** is a PostScript language procedure and some caution is required during creation. Because of the way procedures are created on the operand stack, there is a limit to the amount of in-line data that can be defined. For instance, on a Level 1 device, trying to define a procedure with more than 500 PostScript language objects will overflow the operand stack. Two methods of working around this limitation are discussed here.

One method is to have the application break the **PaintProc** into several sub-procedures, p1, p2, and so on, defined in the form dictionary. Each contains less than 400 PostScript objects, 500 minus some number that may already be on the stack. As shown below, the **PaintProc** itself can then call the sub-procedures.

```
/p1 {...} bind def
/p2 {...} bind def
/p3 {...} bind def
...
/PaintProc {
  begin % put form dict on dict stack
    p1 p2 p3 ...
  end
} bind def
```

This method is fine for hand-converting existing files into forms but is not practical for most applications that generate PostScript language code. Therefore, a modification is necessary. The following code uses the same concept of breaking the **PaintProc** into sub-procedures but allows on-the-fly creation by an application.

```
/PaintProc {{ ...} exec {...} exec {...} exec } bind def
```

Again, each sub-procedure is less than 400 objects. Note that the contents of each sub-procedure should consist of only complete PostScript language objects. For example, within a given procedure, the operators that form strings, arrays, and so forth ( [ ], { }, <<, <, <=, and ), should be succeeded by the corresponding operator that completes construction of the object ( ], } , >> , ~> , and ).

**Note**  The construction operators might be encountered while converting existing files (that is, EPSF) into forms, however, they should be avoided when creating new forms. See section 2, “Example Form Definition,” for more details.

The following method has some execution time overhead but can be useful when converting existing PostScript language files into a **PaintProc** for a Level 2 form.
The following \texttt{bp} (break procedure) procedure, takes a \texttt{name} as a parameter and PostScript language code as data. Every 300 objects of the PostScript language code are condensed into one PostScript language object, an executable array. Then, the \texttt{def} operator is used to associate the condensed code with the \texttt{name} parameter (\texttt{PaintProc} in this case). This allows large amounts of PostScript language code to be placed into a procedure without overflowing the operand stack.

```
/bp { % /ProcName bp --
  mark
  {
    currentfile token not {
      (Error while constructing proc) = pstack flush stop
    } if
    dup /ep eq
    {
      pop
      counttomark array astore cvx bind /exec load ]cvx def
      exit
    } {
      counttomark 300 gt {
        counttomark array astore cvx bind /exec load
      } if
    } ifelse
  } loop
  ) bind def
```

The \texttt{bp} procedure takes a literal name on the operand stack. \texttt{bp} will read from the current file until it encounters the characters “ep.” The procedure will take all the PostScript language tokens read from the current file and build a series of executable arrays of 300 tokens or less. These executable arrays are invoked from one PostScript language procedure that associated with the name passed in on the operand stack. This is done as follows:

```
/PaintProc bp
  % ...code to break up goes here ...
  ep
```

To execute \texttt{bp}, pass it the procedure name follow \texttt{bp} with the code to be condensed and associated with the \texttt{name}. Finally, follow the code with the characters “ep” as shown below:

```
/PaintProc bp
  % code to condense goes here, between bp and ep
  ep
```

\textbf{Note} Note that \texttt{bp} will not handle either \texttt{exec} data or “in line” data being read from the input stream such as that commonly used for large, scanned images following a call to the \texttt{image} operator. Furthermore, reading data directly from the standard input is not suitable for a \texttt{PaintProc} since the \texttt{PaintProc} and form dictionary must be self-contained. See section 4 for information on using images and external devices for storing large amounts of image data.
4 Applications Using Forms

Generally, there are two broad categories of form types implemented as PostScript Level 2 forms. The first category includes forms used multiple times per page or per document, similar to the preceding `totFm` example. We refer to this type as a “stamp” since a given form is stamped several times over a page or throughout a document. The other type is a complex form that can occur only once per document. This form will typically be cached across jobs and may have data overlaid on the form. Examples of this are the forms that are produced by forms management software.

This software allows the production, editing, and printing of the form, as well as merging of data into the form. The two categories overlap somewhat in both directions. For example, the size number on the clothing tag of the `totFm` need not be part of the form. The size numbers can instead be editable data and be imaged separately making this a more practical application.

The forms management software category is a superset of the “stamping” software category, and the information in section 4.1, “Form Stamping Applications,” also applies to section 4.2, “Forms Management Software.”

4.1 Form Stamping Applications

Applications that might use the Level 2 forms mechanism for stamping graphics (such as the `totFm` example) include:

- graphics applications
- page layout applications
- presentation graphics applications
- any application that supports copy and paste functions or duplication of objects

Applications that allow duplication or stamping of graphics objects are ideal for Level 2 forms. If an application does not currently support a user interface for creating stamps of graphical objects, then the underlying data structures and user interface will need to be implemented from scratch.

Applications that provide an interface for defining stamps or duplicating graphical objects have part of what is needed to implement forms. However, most applications do not implement stamps or copies of objects as PostScript language procedures that can simply be turned into a `PaintProc`. They generally produce in-line code for each copy, thus increasing file size and related overhead.
Additional Considerations

- If the user is allowed to edit an instance of a stamp, then the application must be aware that the changed object is no longer an instance of the same stamp.

- If the form is allowed to remain in the form cache across jobs, then an interface for providing an XUID is required. (See the next section, “Caching Forms Across Jobs.”)

Caching Forms Across Jobs

In the previous definition of the totFm form, the form remained in the form cache for the duration of the job. The form, however, had to be executed and placed into the PostScript language form cache by each job that used the form. Caching a form across jobs means that the cached description of the form continues to reside in the printer’s memory across jobs. This is only useful, however, if the form has a name and can be referenced by following jobs. This would mean that the form would need to have been defined outside the serverloop or is accessible via named resources on a disk, for example.

Extended Unique ID

By providing the form with a unique identification number called an extended unique ID (XUID), the PostScript interpreter will cache the form across jobs. This means that the form remains in the cache after job execution thus eliminating the need to re-cache the form during the first execution of execform in subsequent jobs. This method is useful if the form is repeatedly executed across jobs such as during proofing or including a logo in documents. Remember, however, that the form will have to defined in a location that is accessible across jobs, as well.

An XUID is an array of integers whose first value is the organization ID number. The organization ID value of 1000000 is reserved for private interchange in closed environments (forms that will not be distributed to the general public). Closed forms environments should use this value as the starting XUID number. Since XUID arrays starting with 1000000 can be of any length, closed environment will not have the problem of running out of ID space. The administrator of the forms environment should assign second and subsequent elements of the XUID. Suggestions for numbers following the organization number might be a group ID followed by the user’s ID followed by a date and time stamp, and so on.

If a form’s use is not confined to a closed environment, the organization ID number of the XUID array must be assigned by the Adobe™ registry, to avoid caching conflicts. To reserve an organization ID, call the Adobe Systems Developer Association. The remaining elements are controlled by the organi-

**Images**

Image data being read from the standard input is not suitable for forms since the form must be self-contained and available for execution at any time. Data can, however, be read from a printer storage device if one is available. Disk based forms can be managed easily through the Level 2 resource mechanism. For information and an example of using resources, see “Forms as Named Resources,” in section 4.2.

Also note that at the expense of memory, image data can be stored in strings. However, use caution when working with large amounts of data. For example, Level 1 devices have a limit on string length that must be considered and all devices have a finite amount of memory available.

### 4.2 Forms Management Software

Forms management software allows creating, editing, and printing forms, and overlaying data onto the form.

Be aware of the distinction between the PostScript language Level 2 form mechanism as described in the PostScript Language Reference Manual, Second Edition and forms as perceived by an editable “forms” application package available to end users to create and edit forms for data entry use. The level 2 form mechanism does not address the problem of editable form interchange format. However, a forms management application can certainly take advantage of the PostScript Level 2 forms feature.

#### Forms as Named Resources

Level 2 has a built in resource mechanism that can be used to automatically load resources from a printer’s external device into memory. During execution, a PostScript language program can request a form by name. The interpreter loads the form into memory on demand from an external source, such as a disk file, a ROM cartridge, or a network file server.

There are five operators that apply to resources: **findresource**, **resourcestatus**, **resourceforall**, **defineresource**, and **undefineresource**. Note that **defineresource** and **findresource** are similar to the less general operators **definefont** and **findfont**, which are implemented on both Level 1 and Level 2 devices.

The **findresource** operator is the key feature of the resource facility. Given a resource category name (for example, /Form) and an instance name (for example, /totFm), **findresource** returns the form dictionary object if it exists.
If the requested form instance does not exist in memory, **findresource** gets it from an external source and loads it into memory. Similar to using **findfont** for finding and loading fonts into memory, a PostScript language program can access named resources using **findresource** without knowing whether or not they are in memory or how they are obtained from external storage.

**Note** Because of the way **resourcestatus** retrieves information about a resource “size,” a resource must follow the document structuring conventions (DSC) if the size feature of **resourcestatus** is to be supported. In particular, the `%%VMUsage` comment must be present. Also note that “size” in this case does not refer to the size of the cache required but to the amount of memory that the resource requires.

The forms management software must provide for downloading the form resource to the printer hard disk. The following PostScript language code accomplishes the downloading:

```postscript
/buff 128 string def
/fd (Resource/Form/BoxForm) (w) file def
{/loop
  currentfile buff readstring { %ifelse
    fd exch writestring
  }{ %else
    dup length 0 gt { %ifelse
      fd exch writestring
    }{ %else
      pop
    } ifelse
    fd closefile
  } ifelse
} bind
loop

%!PS-Adobe-3.0 Resource-Form
%%Title: BoxForm
%%VMusage: 1432 464
%%BoundingBox: -5 -5 55 55
%%LanguageLevel: 2
5 dict begin
/FormType 1 def
/BBox [ -5 -5 55 55 ] def
/Matrix [1 0 0 1 0 0] def
/PaintProc
{
  pop % pop the form dict, don’t need it
  0 0 moveto 0 50 rlineto 50 0 rlineto 0 -50 rlineto closepath
  gsave .5 setgray fill grestore 5 setlinewidth stroke
} bind def % End PaintProc Code
currentdict % put form dict on top of op stack
end % end form dict
/BoxForm exch /Form defineresource
%%EOF
```
Now, when a **findresource** is executed on /BoxForm, the form will be loaded into memory and the **defineresource** operator will be executed. Note that the code to be downloaded must immediately follow the **loop** token with no comments embedded before the `%!PS-Adobe` comment.

The following is an example of executing a **findresource** to obtain a form dictionary object and then use the object as a parameter to the **execform** operator.

```
/BoxForm /Form findresource % load and define it.
% Stack: dict
/FM1 exch def % save as form number 1, FM1
20 20 translate % position it
FM1 execform % image it
0 60 translate
FM1 execform
showpage
```

**Forms Outside the Server Loop**

This is a method of keeping a form semi-permanently available so that the form definition does not have to be downloaded for each job that executes it. This method is to place the form definition in PostScript VM outside the job server loop. The form remains available until the printer is re-booted at which time the form must be downloaded again.

There are several issues to consider when implementing this method. First, because the form is stored in memory, the memory used by the form is permanently “not available” for general use. Additionally, an application must be able to detect whether or not the form is present. This requires either bidirectional communications, always including the form with the job and letting a spooler strip it out for efficiency, or placing the burden on the user, which becomes a documentation issue.

When defining a form outside the server loop, you may want to consider also assigning the form an XUID, to enable caching across jobs, as well. See the section “Caching Forms Across Jobs” for more information.

The following line of PostScript language code exits in the server loop. Any subsequent PostScript language definitions during the same job become available to ensuing jobs.

```
serverdict begin 0 exitserver
```
5 Form Coordinate Space

Note that the \texttt{BBox} and \texttt{Matrix} entries in the form dictionary are in form space and not default PostScript language user space (as in the case of the DSC \texttt{%%BoundingBox}: comment). For those familiar with PostScript language font definitions, this is similar to the \texttt{FontMatrix} and the \texttt{FontBBox} entries in the font dictionary.

If the \texttt{identmatrix} \[1 \ 0 \ 0 \ 1 \ 0 \ 0\] is used as the \texttt{Matrix} entry of the form dictionary, then the form definition and the corresponding \texttt{BBox} will be in the default PostScript language coordinate space. If it is desired to work in a coordinate space other than the default, take care when creating \texttt{%%BoundingBox}: comments. That is, the form’s \texttt{Matrix} and \texttt{BBox} information must be properly considered and transformed into default user space when creating the \texttt{%%BoundingBox}: comment.

6 Managing the Cache Size

Level 2 devices have adjustable cache sizes for various features including fonts, patterns, and forms. There are two form-related cache values that can be programmatically adjusted.

\texttt{MaxFormItem} is a user parameter stored in the userparams dictionary, consequently, its alteration is subject to \texttt{save} and \texttt{restore}. \texttt{MaxFormItem} specifies the maximum number of bytes a single form can occupy and should be accessed using the \texttt{setuserparams} and \texttt{currentuserparams} operators.

\texttt{MaxFormCache} is a system parameter stored in the systemparams dictionary, which specifies the maximum number of bytes the total combined form cache can occupy. Alterations to this system parameter have a permanent system-wide effect and should not be altered by end user applications.

Closed form environments, however, may have an administrator who can determine the appropriate \texttt{MaxFormCache} value and adjust it accordingly using \texttt{setsystemparams}. Additionally, there will be the need for utilities that allow advanced user inquiry and adjustment of the cache size.

\texttt{CurFormCache} is a system parameter that contains the number of bytes currently occupied by the forms cache.

Note that the default value for \texttt{MaxFormCache} is implementation-dependent and will vary across products.
6.1 Determining a Form's Cache Size

There might be times when an administrator or form creator wants to determine a form's cache requirements on a given device. The following steps can be used to accomplish this.

1. Set MaxFormItem equal to MaxFormCache.

2. Check the number of bytes currently occupied by the total form cache.

3. Execute the new form.

4. Check the number of bytes currently occupied by the total form cache.

5. Subtract the results of step 2 from the results of step 4.

6. Add a 10% buffer to be safe.

The following code fragment implements the above six steps. The last line calculates 10 percent, adds the 10 percent to the form cache size, and then converts the value to an integer for use with the interpreter parameter operators.

```
/max currentsystemparams /MaxFormCache get def
<< /MaxFormItem max >> setuserparams
currentsystemparams /CurFormCache get
formName execform
currentsystemparams /CurFormCache get
sub
1.1 mul cvi
```

In determining the cache size, use caution if the form contains an XUID and is cached across jobs or if the form is executed more than once during the job determining the cache size. The results will be wrong if the form has been cached previously. This problem can be avoided by flushing the cache before each test. The flushing can be achieved by setting the cache to a smaller size than the current size, then re-setting it to a size that will easily accommodate the form being tested. The following code fragment can be used to clear the cache:

```
...
<< /MaxFormCache 0 /Password (XXX) >> setsystemparams
<< /MaxFormCache 200000 /Password (XXX) >> setsystemparams
... now test the form ...
```

where XXX is the system password.
7 Converting Existing Files to Forms

Converting an existing EPS file or a similar stand-alone PostScript language file into a form is quite simple. The steps involved are

1. Create a form dictionary and set the FormType to 1.

2. Set the Matrix entry to the default matrix – [1 0 0 1 0 0].

3. Set the BBox entry equal to the file %BoundingBox values.


5. Use the form by calling execform.

Converting the PostScript language code into a PaintProc is the difficult step. Remember, if the code is more than 500 objects, it will cause a stack overflow on Level 1 devices. Section 3.2, “Working With a Large PaintProc,” offers solutions to this problem.

Also, as mentioned in section 2, “Example Form Definition,” memory for composite objects should not be allocated in the PaintProc or in any procedure called by the PaintProc because it will result in “loss of memory” each time the form definition is executed. This should be considered when converting arbitrary files into forms.

Finally, if converting EPS files into forms is common practice in your environment, it would be wise to write a utility to do the work since converting by hand can be quite time consuming.

8 Limitations

Taking advantage of Level 2 forms generally speeds up rasterization considerably. However, if the form is simple (for example, one or two fonts and some lines), a form not implemented using the Level 2 forms mechanism may already print at the rated engine speed of some desktop printers because of the efficient nature of PostScript language character caching and line drawing. Nevertheless, there are still good reasons for implementing forms using Level 2 forms.

The main reason is that there are now several higher engine speed printers available, and both the number of these fast printers and the speed at which they print is steadily increasing. Just because a form prints at rated engine speed on a 4 or 8 page per minute (ppm) printer, does not mean it will keep up with a printer rated at 12, 20 or 40 ppm. Implementing the form as a PostScript Level 2 form ensures maximum efficiency on all Level 2 printers.
Appendix: Changes Since Earlier Versions

Changes since August 9, 1991 version

• Expanded the section “Caching Forms Across Jobs” in section 4.1, “Form Stamping Applications.”

• Expanded the section “Forms Outside the Server Loop” in section 4.2, “Forms Management Software.”

• Document was reformatted in the new document layout and minor editorial changes were made.
Index

A
applications
form stamping 13

B
bp procedure 12
break procedure. See bp procedure

C
cache size
determining 19
managing 18
CurFormCache 18

D
def 12

E
EF procedure 8
execform 7	emulating 8–12
extended unique ID. See XIUD

F
findresource 15–17
form space 18
forms 5
applications using 13–17
as named resources 15
caching across jobs 14
closed environments 14
common uses 5
converting existing files to 20
definition example 6–7
images 15
limitations 20
memory 17
outside server loop 17
overlaying data onto 10
required dictionary entries 6
forms management software 15
downloading code sample 16

L
Level 2
forms 13

M
Matrix 18
MaxFormCache 18
MaxFormItem 18

P
PaintProc 6, 7
working with a large 11–12
PD procedure 10

R
resourcestatus 16

S
setsystemparams 18
stamp 13
stamping graphics 13

totFm 8, 13
form output 7