Emulation of the makepattern and setpattern Operators

Adobe Developer Support

Technical Note #5112

31 March 1992

Adobe Systems Incorporated
Adobe Developer Technologies
345 Park Avenue
San Jose, CA 95110
http://partners.adobe.com/
## Contents

**Emulation of the makepattern and setpattern Operators**  5

1  Introduction  5

2  Using the Emulation  5
   Definition  6
   Instantiation  7
   Selection and Invocation  7
   Using Color Patterns  9
   Modifying Cache Limits Under Level 2  12

3  Including the Emulation in Your Output  13

4  Emulation Mechanics  15
   Color Patterns  15
   XStep and YStep  16
   TilingTypes  17
   Locking  18
   Known Limitations  19

5  Font Versus Screen Pattern Emulation  19

6  Emulation Code Walkthrough  22
   Level 1 Procedures  22
   Level 2 Veneer Definitions  37

**Appendix: Changes Since Earlier Versions**  41

**Index**  43
1 Introduction

This paper describes the emulation package provided by Adobe Developer Support to allow pattern usage in a consistent manner in both Level 1 and Level 2 printers.

The following items are discussed:

- Basic use of the emulation: How to define, instantiate, select, and invoke a pattern.

- The method in which the emulation or veneer is included in the output stream may depend on the type of target printer or printing environment. Three scenarios are presented.

- The actual mechanics of the Level 1 emulation, including multiple-source patterns, XStep and YStep usage, tiling types, and pattern cell locking are examined. The known limitations of the pattern emulation are also listed.

- A code walk-through of both the Level 1 emulation and the Level 2 veneer.

- A comparison of the font and screen solutions to emulating patterns.

We assumed that the reader is familiar with section 4.9, “Patterns,” of the PostScript Language Reference Manual, Second Edition.

2 Using the Emulation

Painting with a pattern is a four step procedure: definition, instantiation, selection, and invocation. A pattern prototype can be defined once (for example, a simple brick wall pattern), instantiated many times (for example, a brick wall at 0° and 45° with different sized bricks), selected several times, and invoked by painting operators such as stroke, fill, or show.
2.1 Definition

A pattern is defined using a dictionary that contains specific keys describing pattern properties. The required keys in the dictionary are **PatternType**, **PaintType**, **TilingType**, **BBox**, **XStep**, **YStep**, and **PaintProc**. The definition of the pattern is also called the pattern prototype.

Note that the **PaintProc** is the heart of the pattern. Almost any graphics operators can be used to draw the pattern cell. The pattern cell can include graphical elements such as filled shapes, text, and sample images. The shape of the pattern cell does not have to be rectangular.

The following example defines a prototype uncolored pattern dictionary called **Turkey** with a familiar image as the pattern cell.

```latex
10 dict begin
  /PatternType 1 def
  /PaintType 2 def % Uncolored pattern
  /TilingType 1 def % Constant spacing
  /BBox [ 0 0 1 1 ] def % Unit box
  /XStep 1 def
  /YStep 1 def
  /Bitmap <003B00 002700 002480 0E4940 114920 14B220 3CB650 75FE88
    17FF8C 175F14 1C07E2 3803C4 703182 F8EDFC B2BBC2 BB6F84
    31BFC2 18EA3C 0E3E00 07FC00 03F800 1E1800 1FF800> def
  /Matrix [ 24 0 0 -23 0 23 ] def
  /PaintProc {
    begin
      24 23 true Matrix { Bitmap } imagemask
    end
  } bind def
  currentdict
end
/Turkey exch def
```

Note that the size of the pattern cell is 1 point × 1 point, hardly something that can be used directly. When instantiating the pattern, it must be scaled larger.

*Note* The **PaintProc** must not allocate any memory as a result of its execution. Remember that the **PaintProc** can be executed by the pattern machinery many times depending on the pattern complexity and the rendering situation. Any allocation of memory in the **PaintProc** will not be recovered. Each time the **PaintProc** is called more and more memory is consumed. This can lead to a **VMerror**. In addition to avoiding the **def**, **array**, **string**, **dict**, and **matrix** creation operators, also avoid the **[ ], { }, << >>, < >, and <-- ~>** construction operators because they implicitly allocate memory.
2.2 Instantiation

Now that the pattern prototype has been created, the pattern must be instantiated using the \texttt{PA Tmp} emulation procedure (similar to the Level 2 \texttt{makepattern} operator). The \texttt{PA Tmp} procedure takes two operands, a pattern prototype dictionary, and a modifier matrix. The current transformation matrix (CTM) also becomes part of the pattern instance. The size and orientation of the pattern cell is now determined by the CTM at the time of instantiation plus the effects of the modifier matrix. The \texttt{PA Tmp} procedure leaves a pattern instance on the stack that is used as an argument to the \texttt{PATsp} procedure.

The following example rotates the Turkey prototype pattern by $45^\circ$ and scales it to 11 points.

\begin{verbatim}
gsave
  45 rotate
  Turkey [ 11 0 0 11 0 0] Patmp
grestore
/Turkey45 exch def
\end{verbatim}

2.3 Selection and Invocation

Before a pattern can be invoked, it must be selected using the \texttt{PA Tsp} procedure (similar to the Level 2 \texttt{setpattern} operator). This procedure takes a pattern instance (and possibly some color values) as an operand and sets the pattern as the current color.

\textbf{Note} \hspace{1em} This is a convenience procedure. For a more detailed description on the mechanics of pattern selection, see section 4.9 of the “PostScript Language Reference Manual, Second Edition.”

For colored patterns, only the pattern instance must be specified. For uncolored patterns, the color values for either a \texttt{DeviceGray}, \texttt{DeviceRGB}, or \texttt{DeviceCMYK} color space are also needed. When using the pattern package under Level 2, other color spaces are also supported. Note that unlike the \texttt{setpattern} operator, \texttt{PATsp} takes color values in the form of an array.

The following example sets the turkey pattern as the current color and sets the color of the pattern to white. Under Level 2, this also sets the current color space to \texttt{Pattern} along with an underlying color space of \texttt{DeviceGray}.

\begin{verbatim}
[1] Turkey45 PATsp
\end{verbatim}
After the pattern has been selected, all graphical operators that use the current color will paint with that pattern.

```
/Times-Bold findfont 72 scalefont setfont
0 0 moveto
0 setgray
(Adobe) show
[1] Turkey45 PATsp
(Adobe) show
PATusp
```

After a pattern has been used, the driver should issue a `PATusp` (unset pattern) command to paint with normal colored paint.

**Warning** On a Level 1 printer, if a pattern is set inside a `save/restore` pair, it will automatically be unset when the `restore` is executed (as if a `PATusp` had been issued). On a Level 2 printer, the same effect occurs if the `PATsp` appears inside a `save/restore` or a `gsave/grestore`. Additionally, any calls to `setcolor` or `setcolorspace` can unset the current pattern. When in doubt, a driver should set the current pattern using `PATsp` before drawing.

To ensure consistent behavior on both Level 1 and Level 2 platforms, a driver should unset the current pattern before a `grestore, restore`, or any call to change the current color or color space (except when changing the underlying color of an uncolored pattern).

2.4 Using Color Patterns

For performance reasons on Level 1 printers, color patterns can be defined in two formats: regular and multiple-source color patterns. The regular definition will be much slower on Level 1 printers but is more intuitive in its definition.

%! Regular color pattern
7 dict begin
/PatternType 1 def
/PaintType 1 def
/TilingType 1 def
/BBox [-12 -12 12 12 ] def
/XStep 24 def
/YStep 24 def
/PaintProc { % dict
pop
1 0 0 setrgbcolor
clippath fill
0 1 0 setrgbcolor
12 0 moveto 0 0 12 0 360 arc closepath fill
0 0 1 setrgbcolor
0 12 moveto 4 { 144 rotate 0 12 lineto } repeat closepath fill
} def

currentdict
end
gsave
45 rotate
[0.72 0 0 0.72 0 0 ] PATmp
grestore
/CircleStar exch def

Note that the pattern cell is drawn in one execution of the PaintProc. This definition, however, will exhibit poor performance on a Level 1 printer because of the font cache machinery. This is further explained in section 4.1, “Color Patterns.”
The next example definition describes the CircleStar pattern prototype as a multiple-source pattern. There are several new keys in the dictionary: Multi, PaintColors, and PaintData.

%! Multiple source pattern
10 dict begin
/PatternType 1 def
/PaintType 1 def
/TilingType 2 def
/BBox [ -12 -12 12 12 ] def
/XStep 24 def
/YStep 24 def
/Multi 3 def
/PaintColors [ [1 0 0] [0 ] [0.5 0 0.5 0] ] def
/PaintData [ { clippath } bind
{ 12 0 moveto 0 0 12 0 360 arc closepath } bind
{ 0 12 moveto 4 { 144 rotate 0 12 lineto } repeat closepath } bind
] def
/PaintProc { % data dict
pop
exec fill
) def

currentdict
end
gsave
45 rotate
[0.72 0 0 0.72 0 0 ] PATmp
grestore
/CircleStar exch def

Multi

The Multi dictionary entry is a number that determines how PATmp obtains PaintProc data for colored patterns. If Multi is 1 or not present in the dictionary, there is a single pattern source, that is, the pattern supplies its own colors or is uncolored. If Multi is greater than 1, there are multiple pattern sources, that is, there is a PaintData vector that contains the pattern drawing code for each color. The color values for each source are stored in the dictionary entry PaintColors.
PaintColors

The PaintColors dictionary entry is an array of color arrays. There should be one paint color entry for each source in the pattern. Before each source is tiled in the current shape, the color of the source is set by using the appropriate PaintColors entry as an argument to \texttt{PATsc}.

\[
/\text{PaintColors} \[ \[ 0 \] \[ 1 0 0 \] \[ 0.5 0 0.5 0 \] \] \text{ def}
\]

This pattern has three sources. The color of the first source is black, the second is red, and the last is a light green. Color values are allowed in either the DeviceGray, DeviceRGB, or DeviceCMYK color spaces and require 1, 3, and 4 values, respectively.

PaintData

The PaintData dictionary entry is a vector of drawing procedures or data. There should be one entry for each source in the pattern. Depending on which source is being drawn, the corresponding entry in the PaintData array is passed into the PaintProc of the pattern.

\[
/\text{PaintData} \[ \{ \text{clippath} \} \text{ bind}
\{ 12 0 \text{moveto} 0 0 12 0 360 \text{arc closepath} \} \text{ bind}
\{ 0 12 \text{moveto} 4 \{ 144 \text{rotate} 0 12 \text{lineto} \} \text{repeat closepath} \} \text{ bind} \]
\text{ def}
/\text{PaintProc} \{ \% \text{ data dict}
\text{ pop}
\text{ exec fill} \}
\text{ def}
\]

The pattern example in Figure 1 has three sources. The first fills the background, the second draws a circle, and the third draws a star. The PaintProc is executed once for each source in the pattern. Each time, it receives the pattern dictionary and one of the PaintData entries on the stack. It executes the entry (constructing a path) and fills the path.

\textbf{Figure 1} Example of a pattern
Common Errors

The following are three common errors.

- An undefined error occurs during pattern fill. The colors for a multiple-source pattern are set by the pattern machinery, not by the PaintProc. Placing calls to operators that change the current color or color space (for example, setrgbcolor) or calls to the image operator inside either the PaintData array or the PaintProc causes an undefined error to occur on Level 1 printers.

- Because the PostScript™ language imaging model is opaque, there must be one source for each distinct color in the pattern. Do not assume that because a pattern is made of RGB specified colors that only three pattern sources are needed (one each for red, green, and blue). This is not the case.

- Pattern sources obscure each other or are drawn out of order. The pattern is constructed so that the sources are drawn in order: the first entry in the PaintData array corresponds to the first source and is executed first, the second entry in the array is for the second source and is executed second, and so forth. The PostScript language has an opaque imaging model; each source can write over the source below it.

2.5 Modifying Cache Limits Under Level 2

A pattern may or may not be cached depending on the size and complexity of the pattern cell. The pattern cache has two control parameters that determine caching behavior. The MaxPatternItem parameter denotes the maximum number of bytes that can be occupied by a single cached pattern cell; patterns larger than this limit will not be cached. The MaxPatternCache parameter limits the total number of bytes occupied by the pattern cache.

Different devices will have different default values for MaxPatternItem and MaxPatternCache (determined by the OEM). These limits may or may not be sufficient depending on the size of the pattern that is being drawn. To determine the existing cache parameters, issue the following PostScript language code:

```postscript
currentsystemparams /MaxPatternCache get ==
currentuserparams /MaxPatternItem get ==
```

A user application might want to change the MaxPatternItem parameter. The per pattern limit can be changed to m bytes by issuing the following code:

```postscript
<< /MaxPatternItem m >> setuserparams
```
The application should never try to alter the total pattern cache size; `MaxPatternCache` is a systems-level parameter. This is normally done by the systems administrator on a device-by-device basis using the `setsystemparams` operator.

### 3 Including the Emulation in Your Output

There are two parts to the pattern package: a Level 1 emulation and a Level 2 veneer. The emulation includes the font mechanism for tiling with the pattern under Level 1, and the veneer redefines the `makepattern` and `setpattern` operators for Level 2. By using both of these components correctly, a consistent interface for using patterns under Level 1 and Level 2 exists. That is, only the prolog of the document changes; the definition, invocation, and selection of patterns remains the same whether the target printer is Level 1 or Level 2.

If the target printer is a Level 1 device, only the emulation must be sent down with the print job. It should become part of the prolog of the document.

```
%!PS-Adobe-3.0
%%Title: Example of Level 1 output
%%BeginProlog
%%BeginResource: procset Adobe_Patterns_L1 1.0 0
...Level 1 Emulation...
%%EndResource
...other procset definitions...
%%EndProlog
...rest of the document...
%%EOF
```

If the target printer is a Level 2 device, only the veneer must be sent down with the print job. It should become part of the prolog of the document.

```
%!PS-Adobe-3.0
%%Title: Example of Level 2 output
%%LanguageLevel: 2
%%BeginProlog
%%BeginResource: procset Adobe_Patterns_L2 1.0 0
...Level 2 Veneer...
%%EndResource
...other procset definitions...
%%EndProlog
...rest of the document...
%%EOF
```
If the target printer is unknown, such as when a driver is saving to disk, then both the Level 1 emulation and the Level 2 veneer must be sent down with the print job. The PostScript language code in the prolog will configure itself according to the type of printer on which it finds itself. The \texttt{save/restore} pairs are necessary to conserve memory.

%!PS-Adobe-3.0
%!Title: Example of Level 2 output
%!EndComments
%!BeginProlog
/Level2 /languagelevel where {
  pop languagelevel 2 eq { true } { false } ifelse
} { false } ifelse def
/StartLoad {
  dup dup not { userdict /LevelSave save put } if
} bind def
/EndLoad {
  if not { /usedict /LevelSave get restore } if
} bind def
Level 2 not StartLoad {
  %BeginResource: procset Adobe_Patterns_L1 1.0 0
  ...Level 1 Emulation...
  %EndResource
} EndLoad
Level2 StartLoad {
  %BeginResource: procset Adobe_Patterns_L2 1.0 0
  ...Level 2 Veneer...
  %EndResource
} EndLoad
...other procset definitions...
%!EndProlog
...rest of the document...
%!EOF

4 Emulation Mechanics

The pattern emulation uses a Type 3 font as a basis for the pattern. The PaintProc of the pattern prototype dictionary is used as the BuildChar for the font. Because the font machinery includes a built-in cache, speed benefits are realized immediately. The underlying PostScript language code is fairly complex and special redefinitions must be made for each of the marking operators that will use the pattern. For more information on font creation, see the PostScript Language Reference Manual, Second Edition, section 5.7, “Type 3 Fonts.”

Because a Type 3 font allows any PostScript language operator to be used when drawing a character, this emulation allows the drawing of almost any pattern: bitmaps and images, complex drawings, and so forth. The pattern can be rotated, translated, scaled, or skewed at the time of instantiation.

Some of the finer details of the pattern machinery are described in the following sections.

4.1 Color Patterns

Of the two paint types (colored and uncolored), the uncolored pattern is the easiest to emulate. The font machinery is already set up to cache uncolored font bitmaps; therefore, we can treat the pattern as a character in a Type 3 font and obtain the caching benefits.

Because the font cache machinery handles only uncolored bitmaps, we can bypass the font cache for colored patterns using the setcharwidth operator in the BuildChar of the font rather than setcachedevice and allow the font machinery to draw each pattern cell. This functionality is supported in the emulation but makes for a time-consuming pattern fill.

Instead, a new method of defining color patterns is introduced. Each color is laid down as a separate layer in the pattern. The individual color layers are treated as separate uncolored characters in the Type 3 font and are therefore cached. The color for that layer is set before drawing each layer. This type of color pattern is known as a multiple-source pattern.
Support of multiple source patterns in Level 2 is effected through a simple veneer to `makepattern`.

### 4.2 XStep and YStep

The font emulation supports variable inter-cell spacing through the `XStep` and `YStep` variables in the pattern dictionary. This allows you to change the density of the pattern cells in the fill and also allows you to use irregularly shaped patterns. Note that the `XStep` and `YStep` values are specified in reference to the pattern cell coordinate system.

The pattern cells are tiled across the shape with a distance of `XStep` in between each successive cell. Similarly, the pattern cells are vertically tiled with a distance of `YStep` in between each pattern cell.
4.3 TilingTypes

The emulation supports TilingType 1 (constant spacing) only. Using a TilingType of 2 results in no modification of the pattern cell and can lead to stitching problems. TilingType 3 is treated as TilingType 1.

At the time of instantiation for a pattern with a TilingType of 1, the pattern space matrix is determined by taking the CTM and concatenating it with the modifier matrix. Two vectors, [XStep 0] and [0 YStep], are pushed into device space using the dtransform operator and their components rounded to the nearest device pixel. This produces two new vectors \([XS, XS_x]\) and \([YS, YS_y]\) which then have their components divided by XStep and YStep respectively.

These four values replace the \(a, b, c,\) and \(d\) components in the current transformation matrix. The resulting matrix is representative of pattern space. By rounding to pixel boundaries, the pattern cell is slightly distorted (± 1 pixel in both \(x\) and \(y\) directions), but the results ensure a constant spacing of pattern cells that makes the pattern look more consistent.

**Figure 4 Tiling in a snapped pattern space**

In Figure 4, the original pattern cell has a width and height of approximately 7.5 device pixels. This means that the pattern cell might be rendered on the page in the non-uniform manner shown in the first diagram. By using a TilingType of 1, the pattern cell is slightly distorted so that it has a width and height of exactly 8 pixels and the results of the pattern fill are uniform.
4.4 Locking

A pattern cannot be arbitrarily tiled. Tiling a shape with just enough pattern cells to fill the shape is incorrect; the original position of the pattern cell must be taken into account.

**Figure 5** Locking to the pattern cell

In Figure 5, filling the shape with 4×6 tiles would move the tiles out of phase with the position of the pattern cell; the pattern would have been laid down incorrectly. The correct number of tiles to use is 5×7.

Four pieces of information are needed in order to start tiling: the starting position of the tile \((p_x, p_y)\) and the number of tiles to tile across and up \((n_w, n_h)\). PatWidth and PatHeight are the width and height of the pattern cell respectively. The bounding box of the shape to be filled in pattern space is denoted by \((llx, lly, urx, ury)\).

\[
\begin{align*}
p_w &= \left\lfloor llx \div PatWidth \right\rfloor & p_h &= \left\lfloor ur_y \div PatHeight \right\rfloor \\
n_w &= \left\lceil ur_x \div PatWidth \right\rceil - p_w & n_h &= \left\lceil ur_y \div PatHeight \right\rceil - p_h \\
p_x &= p_w \times PatWidth & p_y &= p_h \times PatHeight
\end{align*}
\]
4.5 Known Limitations

The Level 1 emulation closely mirrors the Level 2 implementation of patterns. There are some limitations to the emulation that you should be aware of.

- The emulation supports a TilingType of 1 only. TilingType 3 is considered TilingType 1 in the emulation but without the performance improvement that are found in a Level 2 implementation. It is possible to add TilingType 2 to the emulation, but the calculations needed at pattern fill time make it rather slow.

- Regular color patterns are slow because the font cache does not support colored characters. Multiple-source patterns make up for this deficiency.

- Because only 256 characters can be encoded at any one time in a Type 3 font, only 256 colors are available for use in a multiple-source pattern.

- Patterns cannot be used to fill an imagemask in the Level 1 emulation.

- Stroking complex paths with a pattern is likely to lead to a limitcheck error. This is due to the way in which a path is filled: a strokepath clip is done to establish a clipping region through which the pattern is tiled. A hack in the emulation causes limitcheck errors to be trapped and the path filled with 50% gray, but this is hardly desirable.

- In older versions of the PostScript interpreter, the strokepath operator causes a small amount of memory consumption.

5 Font Versus Screen Pattern Emulation

This paper has discussed a method of implementing patterns known as the pattern font. It is based, in part, on a similar pattern font example presented in section 9.2, “Pattern Fills,” of PostScript Language Program Design. In the past, some applications have implemented patterns using the PostScript language halftone screen machinery known as a pattern screen.

A sample implementation of bitmap pattern fills using pattern screens is discussed in program 15 of the PostScript Language Tutorial and Cookbook. It is important to examine why the pattern font was chosen for this paper rather than pattern screens. The following describe advantages to using pattern screens.

- Color pattern screens are generally defined in device layers: cyan, magenta, yellow, and black allowing for unlimited color usage. A pattern font also allows unlimited colors but at the expense of performance. Multiple-source patterns allow only 256 individual colors.
• A pattern screen is unlikely to generate a limitcheck error during the pattern fill. The font method requires clipping the path which, especially in complex stroking situations, can cause a limitcheck.

• The pattern screen method has a simple prolog and fairly straightforward usage.

The reasons for using a pattern font as an emulation solution are obvious when examining the drawbacks of using a pattern screen.

• A pattern screen is not separable and does not work on devices that do not support halftoning. In addition, a pattern screen will appear at a different size based on the resolution of the output device. This makes the use of pattern screens device-dependent and functionally limited.

• A pattern screen can only support bitmapped patterns. Under Level 2, almost any marking operator can be used to render the pattern cell including text. Only a pattern font can support this functionality.

• Because a consistent interface and maximum functionality for both Level 1 and Level 2 patterns was desired, the pattern emulation had to support different tiling types, arbitrarily scaled and rotated pattern spaces, XStep and YStep spacing, and pattern cell locking. Pattern screens cannot support this functionality without performing mathematical gyrations on either the host or the printer.

It can be argued that performance of the pattern fill outweighs any functional issues. Part of the work done to create the font emulation was a series of tests pitting the screen against the font method for bitmap pattern fills. The bulk of the fill time for using a pattern screen is in setting the pattern. The bulk of the fill time for a pattern font is in the filling itself.

In Figure 6, pattern test a is a full page fill that tests the tiling abilities of the pattern mechanism. Pattern test b fills several elementary shapes with the same pattern. Tiling test c fills a complex shape with a single pattern. Tiling test d fills several elementary shapes with a variety of patterns. This is meant to test not only fill time but switching time between patterns as well.
Figure 6  *Pattern tests*

Table 1  *Timing results*

<table>
<thead>
<tr>
<th>Printer</th>
<th>Method</th>
<th>Full page</th>
<th>Elementary shapes</th>
<th>Complex shapes</th>
<th>Many patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>LaserWriter II NTX</td>
<td>Screen</td>
<td>3.92</td>
<td>2.73</td>
<td>3.67</td>
<td>22.52</td>
</tr>
<tr>
<td></td>
<td>Font</td>
<td>3.18</td>
<td>4.05</td>
<td>17.34</td>
<td>7.24</td>
</tr>
<tr>
<td>DataProducts LZR 2665</td>
<td>Screen</td>
<td>10.03</td>
<td>7.99</td>
<td>9.58</td>
<td>61.48</td>
</tr>
<tr>
<td></td>
<td>Font</td>
<td>5.62</td>
<td>14.42</td>
<td>57.49</td>
<td>16.40</td>
</tr>
<tr>
<td>Linotype L200</td>
<td>Screen</td>
<td>23.96</td>
<td>25.52</td>
<td>25.79</td>
<td>192.99</td>
</tr>
<tr>
<td></td>
<td>Font</td>
<td>1.78</td>
<td>5.20</td>
<td>100.62</td>
<td>10.77</td>
</tr>
<tr>
<td>LaserWriter Plus</td>
<td>Screen</td>
<td>12.24</td>
<td>13.40</td>
<td>11.44</td>
<td>70.29</td>
</tr>
<tr>
<td></td>
<td>Font</td>
<td>34.49</td>
<td>39.22</td>
<td>294.21</td>
<td>46.02</td>
</tr>
<tr>
<td>QMS ColorScript</td>
<td>Screen</td>
<td>13.07</td>
<td>12.48</td>
<td>12.30</td>
<td>55.15</td>
</tr>
<tr>
<td></td>
<td>Font</td>
<td>17.31</td>
<td>12.70</td>
<td>60.20</td>
<td>13.78</td>
</tr>
<tr>
<td>Canon CLC</td>
<td>Screen</td>
<td>12.31</td>
<td>12.09</td>
<td>11.69</td>
<td>58.99</td>
</tr>
<tr>
<td></td>
<td>Font</td>
<td>8.59</td>
<td>4.17</td>
<td>10.68</td>
<td>4.70</td>
</tr>
</tbody>
</table>

For a full page fill, the font method is generally faster than a pattern screen. This is due to the large number of calculations that must be done to initially set up a pattern screen.

When filling elementary shapes, the font pattern is less efficient than using a screen, especially on printers with low end CPUs. On the Linotronic® 200 and the Canon® CLC 500, the pattern font exhibits better performance due to the high setup cost for screens at higher resolutions.
When filling a complex shape, the font pattern is clearly lacking. This is because of the calculations needed to tile the font through the complex clipping path. On newer interpreters, the clipping algorithms have been improved, and the font method performance is similar to the pattern screen.

When switching between many different patterns, the font method is clearly superior. For screens, the halftoning mechanism must be invoked each time the pattern is switched.

In conclusion, it cannot be said that the pattern font is faster than the pattern screen; they perform differently depending on the printing situation. The pattern font can, however, be imbued with more functionality than the screens and was, therefore, the chosen method for the emulation.

6 Emulation Code Walkthrough

The following sections step through all the procedures written to emulate the Level 2 pattern functionality on Level 1 interpreters.

6.1 Level 1 Procedures

\( cx \ cy \ char \ rx \ ry \ string \) \( \text{PATwidthshow} \)

The \( \text{PATwidthshow} \) procedure is similar to the \( \text{widthshow} \) operator, but it fills the character shapes with the current pattern. The redefinition of the show operators (\( \text{show} \), \( \text{ashow} \), \( \text{widthshow} \), and \( \text{awidthshow} \)) use this routine.

Inputs: \( cx \ cy \) – Add these values to the width of \( char \) when showing the string.
\( char \) – Specially treated character, for example, blank.
\( rx \ ry \) – Add these values to the width of all characters (including \( char \)) when showing the string.
\( string \) – The string to be shown.

Used By: \( \text{PATredef} \)

To reduce the chance of a \text{limitcheck} error, the procedure breaks down the requested string into individual characters. Each character is converted to a path and then used as a clip for the resulting fill. If the character is equal to \( char \), the current point is modified by \( cx \) and \( cy \). The current point is always modified by \( rx \) and \( ry \).

```
% This junk string is used by the show operators
/PATstr 1 string def
/PATwidthshow { % cx cy cchar rx ry string
   % Loop over each character in the string
   { % cx cy cchar rx ry char
```
% Show the character
 dup % cx cy cchar rx ry char char
 PATsstr dup 0 4 -1 roll put % cx cy cchar rx ry char (char)
 false charpath % cx cy cchar rx ry char
 /clip load PATdraw

% Move past the character (charpath modified the
% current point)
 currentpoint % cx cy cchar rx ry char x y
 newpath
 moveto % cx cy cchar rx ry char

% Reposition by cx, cy if the character in the
% string is cchar
 3 index eq { % cx cy cchar rx ry
    4 index 4 index rmoveto
  } if

% Reposition all characters by rx ry
 2 copy rmoveto % cx cy cchar rx ry
 } forall
 pop pop pop pop pop % -
 currentpoint
 newpath
 moveto
 } bind def

– PATcg gstatedict

The PATcg procedure is similar but not equivalent to the currentgstate operator in Level 2. It saves the current line width, line cap, line join, and miter limit values, as well as the current dash, color, and the current transformation matrix. These graphics state values are needed at the time the pattern is drawn.

Outputs: gstatedict – A dictionary containing the gstate variables.

Used By: PATmp

/PATcg {
  7 dict dup begin
    /lw currentlinewidth def
    /lc currentlinecap def
    /lj currentlinejoin def
    /ml currentmiterlimit def
    /ds [ currentdash ] def
    /cc [ currentrgbcolor ] def
    /cm matrix currentmatrix def
  end
} bind def
proc PATdraw –

The PATdraw procedure will fill the current path with the current pattern.

Inputs: proc – This should be a type of clipping procedure. PATdraw will clip to the current path with this procedure and fill the clipping region with the current pattern.

Used By: PATredef, PATawidthshow, PATkshow

The number of pattern cells needed to fill the current path in pattern space and the starting position of the fill is determined using the PATpcalc routine. The clipping procedure is then executed, and the resulting clipping region filled.

```
% PATdraw – calculates the boundaries of the object and
% fills it with the current pattern
/PATdraw { % proc
  save exch
  PATpcalc % proc nw nh px py
  5 -1 roll exec % nw nh px py
  newpath
  PATfill % -
  restore
} bind def
```

nw nh px py PATfill –

This routine is the core of the emulation. Given the starting position of the tiling and the number of cells to tile, the routine ‘shows’ a string of source characters in the pattern font across the shape to be filled.

Inputs: nw, nh – The number of pattern cells to paint horizontally and vertically to completely fill the current clipping region.
px, py – The starting position of the first pattern cell to be shown.

Used By: PATdraw, PATstroke

The current pattern font is located in the PATDict dictionary and is set as the current font. The gstate at the time the pattern was instantiated (PATmp) is re-established. This includes setting the CTM to represent pattern space. If an uncolored pattern is being used, the color specified during its selection (PATsp) is set.
% PATfill - performs the tiling for the shape
/PATfill { % nw nh px py PATfill -
    PDict /CurrentPattern get dup begin
    setfont
    % Set the coordinate system to Pattern Space
    PatternGState PATsg

    % Set the color for uncolored patterns
    PaintType 2 eq { PDict /PColor get PATsc } if

    The shape will be filled in as many passes as there are sources in the pattern.
    Each pass starts tiling at position (px, py). If the pattern is a multiple source
    pattern, the color appropriate for that pass is set. The characters in the pattern
    font are organized so that character 0 corresponds to the first pattern source,
    character 1 to the second, and so on. A string is used in the tiling process, and
    its contents are set according to the source being tiled.

    % Create the string for showing
    3 index string % nw nh px py str
    % Loop for each of the pattern sources
    0 1 Multi 1 sub { % nw nh px py str source
        % Move to the starting location
        3 index 3 index % nw nh px py str source px py
        moveto % nw nh px py str source

        % For multiple sources, set the appropriate color
        Multi 1 ne { dup PaintColors exch get PATsc } if

        % Set the appropriate string for the source
        0 1 7 index 1 sub for pop { 2 index exch 2 index put} for pop

    The next loop tiles the shape with the pattern font. The pattern is painted nw
    cells across using the string created at the beginning of the loop and nh cells
    upwards using the show operator.

    % Loop over the number of vertical cells
    3 index % nw nh px py str nh
    { % nw nh px py str
        currentpoint % nw nh px py str cx cy
        2 index show % nw nh px py str cx cy
        YStep add moveto % nw nh px py str
    } repeat % nw nh px py str
} for
5 { pop } repeat
end
} bind def
A demonstration of the looping behavior in PATfill is shown in Figure 7. A multiple-source pattern made up of three colors is used to fill a shape. This involves three passes over the shape. The first lays down the green layer, the second a magenta circle, and the third a blue circle. This is faster than if the pattern font had been defined as colored font. Each source character gets saved in the font cache, reducing fill time.

**Figure 7** A multiple-source, three-color pattern

proc string PATkshow –

The PATkshow procedure is similar to the kshow operator, but it fills the character shapes with the current pattern.

*Inputs: proc* – This procedure is executed for each character in the string except the last one. Both the character being showed and the next character in the string are passed into this routine on the stack.

*string* - The string to be shown.

*Used By:* PATredef
In order to reduce the likelihood of a limitcheck error, the procedure breaks down the requested string into individual characters. Each character is converted to a path and then used as a clip for the resulting fill. The character and its neighbor are placed on the stack, and the user proc is then executed.

```
% PATkshow - kshow with the current pattern
/PATkshow { % proc string
  exch bind % string proc
  1 index 0 get % string proc char
  % Loop over all but the last character in the string
  0 1 4 index length 2 sub {
    % string proc char idx
    % Find the n+1th character in the string
    3 index exch 1 add get % string proc char char+1
    exch 2 copy % string proc char+1 char char+1 char
    % Now show the nth character
    PATsstr dup 0 4 -1 roll put % string proc chr+1 chr chr+1 (chr)
    false charpath % string proc char+1 char char+1 char
    /clip load PATdraw
  } for
  % Now display the last character
  PATsstr dup 0 4 -1 roll put % string proc (char+1)
  false charpath % string proc
  /clip load PATdraw
  newpath
  pop pop % -
} bind def
```

`patdict patmtx PATmp patinstance`

This routine is the equivalent of the PostScript Level 2 makepattern operator. In addition to the uncolored and colored patterns, a multiple-source pattern can also be defined.

**Inputs:** `patdict` – This is the prototype pattern dictionary.  
`modmtx` – This is a matrix that will modify pattern space to produce scaled, rotated, translated, and skewed effects.  
`CTM` – The current graphics state at the time of the `PATmp` call which will affect the pattern instantiation.

**Outputs:** `patinstance` – This is an instance of the prototype pattern modified with respect to the CTM at the time of instantiation, the modifier matrix, and the TilingType of the pattern.
% PATmp - the makepattern equivalent
/PATmp { % patdict patmtx PATmp patinstance
   exch dup length 7 add % We will add 6 new entries plus 1 FID
dict copy % Create a new dictionary
begin
% Matrix to install when painting the pattern
TilingType PATtcalc
/PatternGState PATcg def
PatternGState /cm 3 -1 roll put
% Check for multi pattern sources (Level 1 fast color patterns)
currentdict /Multi known not { /Multi 1 def } if
% Font dictionary definitions
/FontType 3 def
% Create a dummy encoding vector
/Encoding 256 array def
3 string 0 1 255 {
   Encoding exch dup 3 index cvs cvn put } for pop
/FontMatrix matrix def
/FontBBox BBox def
/BuildChar {
   mark 3 1 roll % mark dict char
   exch begin
   Multi 1 ne {PaintData exch get}{pop} ifelse% mark
   [paintdata]
      PaintType 2 eq Multi 1 ne or
      { XStep 0 FontBBox aload pop setcachedevice }
      { XStep 0 setcharwidth } ifelse
   currentdict % mark [paintdata] dict
   /PaintProc load % mark [paintdata] dict paintproc
   end
   gsave
   false PATredef exec true PATredef
   grestore
cleartomark % -
} bind def
}ind def

The following paragraphs explain each section of code in detail.

The pattern needs extra entries in addition to the entries provided by the user if it is to be defined as a font. The supplied pattern dictionary is copied and extra space is allocated. The dictionary is then put on the dictionary stack and the extra entries defined into it.

exch dup length 7 add % We will add 6 new entries plus 1 FID
dict copy % Create a new dictionary
begin
The pattern space in which the pattern will be drawn depends on the modifier matrix, the current graphics state, and the TilingType. The PATtcalc routine uses this information and calculates a pattern matrix that defines pattern space. The current graphics state is then saved in PatternGState with the PATtcalc matrix used as the current transformation matrix (cm).

% Matrix to install when painting the pattern
TilingType PATtcalc
/PatternGState PATcg def
PatternGState /cm 3 -1 roll put

The Multi variable indicates the number of sources in the pattern. If it is not specified by the user then the pattern is considered to have only one source.

% Check for multiple pattern sources (Level 1 fast colour
% patterns)
currentdict /Multi known not { /Multi 1 def } if

Additional definitions are necessary in order to turn this dictionary into a Type 3 font.

% Font dictionary definitions
/FontType 3 def
% Create a dummy encoding vector
(Encoding 256 array def
3 string 0 1 255 {Encoding exch dup 3 index cvs cvn put} for pop
/FontMatrix matrix def
/FontBBox BBox def

The most important portion of the font is defined next: the BuildChar procedure. The BuildChar procedure is called when the pattern is first used, that is, when the character(s) that represent the pattern are shown for the first time. The result of the call to BuildChar is cached in the font cache if there is room in the cache. The BuildChar for the pattern font runs the PaintProc provided by the user.

A mark is placed on the stack for clean-up in the event that the user’s PaintProc forgets to consume the pattern dictionary and possibly the paint data passed in to it. If the pattern is a multiple-source pattern, the paint data is retrieved from the pattern dictionary. The pattern data to retrieve is determined by the character code passed into BuildChar.

If the pattern is uncolored or is a multiple-source pattern, then the setcachedevice operator is used to gain caching benefits. Colored patterns cannot be cached and must use the setcharwidth operator instead. Note that the width of the character in both cases is set to XStep, this effects the XStep inter-cell spacing when showing the character.
The **PaintProc** is then loaded onto the stack and executed (the filling, stroking, and showing routines are set back to their original definitions to avoid recursion). Finally, any leftovers on the stack are removed.

```
/BuildChar { % mark dict char
  mark 3 1 roll
  exch begin
    Multi 1 ne { PaintData exch get } { pop } ifelse
    PaintType 2 eq Multi 1 ne or
      { XStep 0 FontBBox aload pop setcachedevice }
      { XStep 0 setcharwidth } ifelse
    currentdict % mark [paintdata] dict
    /PaintProc load % mark [paintdata] dict paintproc
  end
  gsave
  false PATredef exec true PATredef
  grestore
  cleartomark % -
} bind def
```

The new dictionary is now ready to become a font. It is placed on the operand stack and closed, and the pattern font is defined using **definefont**. The end result on the stack is a Type 3 font dictionary, the pattern instance.

```
currentdict % newdict
/foo exch % /foo newdict
definefont % newfont
} bind def
```

– **PATpcalc** *nw nh px py*

The **PATpcalc** routine calculates the starting point and the width and height of the tile fill for the current path.

**Inputs:** The current path.

**Outputs:** *nw, nh* – The number of pattern cells to paint horizontally and vertically in order to completely fill the current path.

*px, py* – The starting position of the first pattern cell to be shown.

**Used By:** **PATfill**

The tiling starting point, width, and height, are determined in pattern space, so the matrix at the time of pattern instantiation is set. Additionally, if the pattern is not drawn at the origin, a translation to the lower left corner of the bounding box of the pattern cell is necessary. This ensures that the pattern is locked down (tiled with respect to its original position). The bounding box of the path is obtained \((ll_x ll_y ur_x ur_y)\), and the output values calculated \((n_w n_h p_x p_y)\). See section 4.4 for the formulas used to calculate these values.
% PATpcalc - calculates the starting point and width/height
% of the tile fill for the shape
/PATpcalc { % - PATpcalc nw nh px py
    PATdict /CurrentPattern get begin
    gsave
    % Set up the coordinate system to Pattern Space
    % and lock down pattern
    PatternGState /cm get setmatrix
    BBox aload pop pop pop translate
    % Determine the bounding box of the shape
    pathbbox % llx lly urx ury
    grestore
    % Determine (nw, nh) the # of cells to paint width and height
    PatHeight div ceiling % llx lly urx qh
    4 1 roll % qh llx lly urx
    PatWidth div ceiling % qh llx lly qw
    4 1 roll % qw qh lly ph
    PatHeight div floor % qw qh 1lx ph
    4 1 roll % ph qw ph 1lx
    PatWidth div floor % ph qw ph pw
    4 1 roll % pw ph qw ph
    2 index sub cvi abs % pw ph qw ph
    exch 3 index sub cvi abs exch % pw ph nw-qw-pw nh=qh-ph

    % Determine the starting point of the pattern fill
    %(px, py)
    4 2 roll % nw nh pw ph
    PatHeight mul % nw nh pw py
    exch % nw nh py pw
    PatWidth mul exch % nw nh px py
    end
} bind def

bool PATredef –

The PATredef procedure redefines the standard filling, stroking, and showing procedures to either use the pattern emulation or not. After the procedure is defined, it is executed with a false argument so that the operators are redefined to procedures containing the old definitions. This is to prevent future procedures in the prolog from binding in the true operators and preventing their redefinition when patterns are used.

Inputs: If bool is true, the operators are redefined to use patterns, if bool is false, the operators revert to their normal functionality.

Used By: BuildChar, PATsp, PATusp
% Save the original routines so that we can use them later on
/oldfill /fill load def
/oldeofill /eofill load def
/oldstroke /stroke load def
/oldshow /show load def
/oldashow /ashow load def
/oldwidthshow /widthshow load def
/oldawidthshow /awidthshow load def
/oldkshow /kshow load def

% These defs are necessary so that subsequent procs don't bind in % the originals
/fill { oldfill } bind def
eofill { oldeofill } bind def
/stroke { oldstroke } bind def
/show { oldshow } bind def
/ashow { oldashow } bind def
/widthshow { oldwidthshow } bind def
/awidthshow { oldawidthshow } bind def
/kshow { oldkshow } bind def

/PATredef {
  userdict begin
    {
      /fill { /clip load PATdraw newpath } bind def
      /eofill { /eoclip load PATdraw newpath } bind def
      /stroke { PATstroke } bind def
      /show { 0 0 null 0 0 6 -1 roll PATawidthshow } bind def
      /ashow { 0 0 null 6 3 roll PATawidthshow }
        bind def
      /widthshow { 0 0 3 -1 roll PATawidthshow }
        bind def
      /awidthshow {PATawidthshow } bind def
      /kshow { PATkshow } bind def
    }
  /fill { oldfill } bind def
eofill { oldeofill } bind def
/stroke { oldstroke } bind def
/show { oldshow } bind def
/ashow { oldashow } bind def
/widthshow { oldwidthshow } bind def
/awidthshow { oldawidthshow } bind def
/kshow { oldkshow } bind def
} ifelse
end
} bind def
false PATredef
[comp₁, ..., compₙ] PATsc –

The PATsc procedure is similar but not equivalent to the Level 2 setcolor operator. It sets the current color according to the number of components in the component array.

**Inputs:** components – Instead of loose color values as arguments, PATsc takes an array containing the color values for the current color space. Colors are specified in either the DeviceGray, DeviceRGB, or DeviceCMYK color spaces (arrays of 1, 3, and 4 numbers respectively) when using this procedure.

**Used By:** PATfill

```latex
% Conditionally define setcmykcolor if not available.
/setcmykcolor where { pop } {
    /setcmykcolor {
        3 { 3 index add neg dup 0 lt { pop 0 } if 3 1 roll } repeat
        setrgbcolor pop
    } bind def
} ifelse

/PATsc { % colorarray
   aload length % c₁ ... cₙ length
    dup 1 eq { pop setgray } { 3 eq { setrgbcolor } { setcmykcolor } ifelse } ifelse
} bind def
```

gstatedict PATsg –

The PATsg is similar but not equivalent to the setgstate operator in Level 2. It sets the current line width, line cap, line join, and miter limit values, as well as the current dash, color, and the current transformation matrix. These values are set at the time the pattern is drawn.

**Inputs:** gstatedict – A dictionary containing the gstate variables.

**Used By:** PATfill

```latex
/PATsg { % dict
    begin
        lw setlinewidth
        lc setlinecap
        lj setlinejoin
        ml setmiterlimit
        ds aload pop setdash
        cc aload pop setrgbcolor
        cm setmatrix
    end
} bind def
```
The PATsp procedure is the equivalent to the Level 2 setpattern operator which establishes pattern as the current color space and the specified pattern as the current color. This emulation of setpattern takes the pattern instance as an argument; for uncolored patterns, the color components of the pattern should be specified as well.

Inputs: pattern – This is a pattern instance dictionary.
components – This array contains the color components of the color to be used when drawing with the uncolored (PaintType = 2) pattern.

PATsp first redefines the basic filling, stroking, and showing operators so that they will draw with patterns. A scratch dictionary, PATDict, is used to hold the current pattern, its user specified color (uncolored patterns only), and the current color.

```
/PATDict 3 dict def

/PATsp { true PATredef
     PATDict begin
     /CurrentPattern exch def
     % If it's an uncolored pattern, save the color
     CurrentPattern /PaintType get 2 eq {
         /PColor exch def
     } if
     /CColor [ currentrgbcolor] def
     end
 } bind def
```

– PATstroke –

The PATstroke procedure strokes the current path with the current pattern. This is done using the strokepath operator to create a shape suitable for clipping and then tiling the pattern cell across the clipping path.

The strokepath operator can produce a large and complex path, especially when many curves are in the original path. This can result in a limitcheck error. The emulation compensates for this by executing the pattern fill in a stopped context so that error recovery can occur. If the current path cannot be stroked with the current pattern, it is stroked with 50% gray instead.

Inputs: The current path.

Used By: PATredef
% PATstroke - stroke with the current pattern
/PATstroke { 
  countdictstack 
  save 
  mark 
  { 
    currentpoint strokepath moveto 
    PATpcalc % proc nw nh px py 
    clip newpath PATfill 
  } stopped { 
    (** PATstroke Warning: Path is too complex, stroking 
    with gray) = 
    cleartomark 
    restore 
    countdictstack exch sub dup 0 gt 
    { { end } repeat } { pop } ifelse 
    gsave 0.5 setgray oldstroke grestore 
  } { pop restore pop } ifelse 
  newpath 
} bind def

modmtx tilingtype PATtcalc tilemtx

When filling the requested shape with a pattern, the pattern is drawn in pattern space. Pattern space can be defined using a special matrix and the setmatrix operator. This matrix is stored in the PatternMatrix field of the pattern font. The PATtcalc routine calculates this matrix.

**Inputs:** modmtx – This is the modifier matrix. It is concatenated with the current graphics state in order to modify the pattern prototype.

  tilingtype - either 1, 2, or 3. This is the TilingType of the pattern.

**Outputs:** tilemtx – This matrix defines pattern space.

**Used By:** PATmp

The pattern space in which the pattern will be drawn depends on the modifier matrix, the current graphics state, and the TilingType. The modifier matrix is first concatenated to the current transformation matrix; the CTM is now set up to represent the undistorted pattern space. For TilingTypes 1 and 3, the pattern space must be slightly distorted to ensure that the pattern occupies an integral number of device pixels. This is accomplished by pushing the XStep and YStep values into device space and rounding them to the nearest pixel.

Neither TilingType 2 nor the faster TilingType 3 is supported in this emulation.
36 Emulation of the makepattern and setpattern Operators (31 Mar 92)

/PATtcalc {  \% modmtx tilingtype PATtcalc tilematrix
  \% Note: tiling types 2 and 3 are not supported.
  gsave
    exch concat \% tilingtype
    matrix currentmatrix exch \% cmtx tilingtype
    \% Tiling type 1 and 3: constant spacing
    2 ne { 
      \% Distort the pattern so that it occupies
      \% an integral number of device pixels
      dup 4 get exch dup 5 get exch \% tx ty cmtx
      XStep 0 dtransform
      round exch round exch \% tx ty cmtx dx.x dx.y
      XStep div exch XStep div exch \% tx ty cmtx a b
      0 YStep dtransform
      round exch round exch \% tx ty cmtx a b dy.x dy.y
      YStep div exch YStep div exch \% tx ty cmtx a b c d
      7 -3 roll astore \% [ a b c d tx ty ]
    } if
  grestore
} bind def

–PATusp–

Under Level 2, the setting of a pattern is considered part of the graphics state. Therefore, it obeys gsave/grestore and save/restore effects and can be affected by a variety of operators including color and color-space operators (setcolor, setcolorspace, setgray, setrgbcolor, and so on), initgraphics, and others. The emulation obeys the save/restore only and is not affected by other changes to the graphics state. Thus, the PATusp procedure is necessary to allow the unsetting of a pattern. It sets the filling, stroking, and showing operators back to their old definitions and restores the current color.

/PATusp {  
  false PATredef
  PATDict begin
    CColor PATsc
  end
} bind def
6.2 Level 2 Veneer Definitions

Under Level 2, several veneers have been created so that patterns can be defined, instantiated, and selected with the same syntax whether the target printer is Level 1 or Level 2.

\textit{patdict patmtx PATmp patinstance}

This veneer uses the PostScript Level 2 \texttt{makepattern} operator. In addition to the uncolored and colored patterns, a multiple-source pattern can also be defined. A multiple-source pattern is redrawn by the \texttt{PATmp} veneer as a normal color pattern, unlike the Level 1 emulation, the Level 2 pattern machinery caches color patterns.

\textit{Inputs: patdict} – This is the prototype pattern dictionary.  
\textit{modmtx} – This matrix will modify pattern space to produce scaled, rotated, translated, and skewed effects.  
\textit{CTM} – The current graphics state at the time of the \texttt{PATmp} call will affect the pattern instantiation.

\textit{Outputs: patinstance} – An instance of the prototype pattern modified with respect to the CTM at the time of instantiation, the modifier matrix, and the \texttt{TilingType} of the pattern.

```
% PATmp - makepattern veneer
/PATmp { % patdict patmtx PATmp -
    exch dup length 2 add dict copy
    begin
      currentdict /Multi known not { /Multi 1 def } if
      % Special case for multiple sources
      Multi 1 ne {
        /UserProc /PaintProc load def
        /PaintProc {
          begin
            0 1 Multi 1 sub { % idx
              PaintColors 1 index get PATsc % idx
              PaintData exch get % pdatum
              gsave
              currentdict % pdatum dict
              UserProc
              grestore
            } for
            end
          } bind def
        } if
      currentdict
      end
      exch makepattern
    } bind def
```
\( \text{[comp, ... comp,]} \) \text{PATsc} –

The \text{PATsc} procedure is similar but not equivalent to the Level 2 \text{setcolor} operator. It sets the current color according to the number of components in the component array.

\text{Inputs: components} – Instead of loose color values as arguments, \text{PATsc} takes an array containing the color values for the current color space. Colors are specified in either the \text{DeviceGray}, \text{DeviceRGB}, or \text{DeviceCMYK} color spaces (arrays of 1, 3, and 4 numbers respectively) when using this procedure.

\text{Used By: PATtmp}

\%
\text{PATsc – setcolor equivalent (takes an array as a parameter instead of loose values)}
/\text{PATsc} { % \text{colorarray}
  \text{mark exch aload pop}
  \text{counttomark 1 eq } \{ \text{setgray} \} \text{ if}
  \text{counttomark 3 eq } \{ \text{setrgbcolor} \} \text{ if}
  \text{counttomark 4 eq } \{ \text{setcmykcolor} \} \text{ if}
  \text{counttomark 0 ne } \{ \text{setcolor} \} \text{ if}
  \text{cleartomark}
} \text{ bind def}

\text{pattern PATsp} –
\text{[comp1 ... compn] pattern PATsp} –

The \text{PATsp} veneer uses the Level 2 \text{setpattern} operator, which establishes \text{pattern} as the current color space and the specified pattern as the current color.

\text{Inputs: pattern} – This is a pattern instance dictionary.
\text{components} – This array contains the color components of the color to be used when drawing with the uncolored (\text{PaintType} = 2) pattern.

\text{PATsp} first saves the current color and color space so that it can be re-established by \text{PATusp}. If the pattern is uncolored, \text{PATsp} allows the specification of colors in any of the device color spaces and sets the underlying color space of the pattern appropriately. Finally, \text{setpattern} is called to establish the pattern as the current color.
% setpattern veneer
/PATsp { % [colors...] pattern PATsp -
PATDict begin
  /CColor [ currentcolor ] def
  /CCSpace currentcolorspace def
end
dup /PaintType get 2 eq { % Uncolored pattern
  % Allow quick 'n easy rotation between the device colorspaces.
  exch dup length % pattern [colors...] clen
dup 1 eq { [/Pattern /DeviceGray] setcolorspace } if
  dup 3 eq { [/Pattern /DeviceRGB] setcolorspace } if
  4 eq { [/Pattern /DeviceCMYK] setcolorspace } if
 aload length 1 add -1 roll % comp1, comp2, ... compn dict
} if
setpattern
} bind def

-- PATusp --

Under Level 2, setting a pattern is considered part of the graphics state. It, therefore, obeys gsave/grestore and save/restore effects and can be affected by a variety of operators including color and color-space operators (setcolor, setcolorspace, setgray, setrgbcolor, and so on), initgraphics, and others. The PATusp veneer is a convenience; it simply resets the color space and color that existed before the call to PATsp.

% unsetpattern routine
/PATusp { PATDict begin
  CCSpace setcolorspace
  CColor aload pop setcolor
end
} bind def
Appendix: Changes Since Earlier Versions

Changes since August 5, 1991 version

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

Changes since May 4, 1991 version

• A note about memory leaks inside the PaintProc was added to section 2.1, “Definition.”

• The section 2.5, “Modifying Cache Limits Under Level 2,” was added.
## Index

### A, B
BuildChar 15, 29, 31

### C
CircleStar, example 10
- color patterns
  - regular definition 9
  - using 9–12
- components 33, 34, 38
CTM. See current transformation matrix
- current transformation matrix 7

### D
definefont 30
- DeviceCMYK 7
- DeviceGray 7
- DeviceRGB 7
dtransform 17

### E
emulations
- code walkthrough 22
- font versus screen pattern 19–22
- including in output 13
- limitations 19
- mechanics 15–19
- using 5–13

### F
font method 21
- font pattern 21

### G, H
grestore 8
gstatedict 23, 33

### I, J, K
imagemask 19
- initgraphics 36

### L
- Level 2
  - cache limits
    - modifying 12
  - limitcheck error 19, 20
  - locking
e  - to a pattern cell 18

### M, N
makepattern 16, 27, 37
- MaxPatternCache 12
- MaxPatternItem 12
- modmtx 27, 35, 37
- Multi 29
- Multi dictionary 10
- multiple-source pattern
  - CircleStar 10

### O
- operators
  - color space 36

### P, Q
- PaintColors dictionary 11
- PaintData 10, 11
- PaintData dictionary 11
PaintProc  6, 9, 11, 15, 29
 error  6
PATawidthshow  22
PATcg  23
PATDict  24, 34
patdict  27, 37
PATdraw  24
PATfill  24, 30, 33
PatHeight  18
patinstance  27, 37
PATkshow  26
PATmp  7, 27, 35, 38
PATpcalc  30
PATredef  22, 26, 31, 34
PATsc  11, 33
PATsp  31, 34
PATstroke  24, 34
PATcalc  30
PATsg  33
PATtdraw  24
PATtcalc  29, 35
pattern   34
pattern cell
 size  6
pattern dictionary
 turkey example  6
 uncolored example  6
pattern fill
 undefined error  12
pattern font  19
pattern prototype  6
pattern screen  19
drawbacks  20
pattern sources
 error  12
PatternGState  29
PatternMatrix  35
patterns
 colored  15
 multiple source  26
 multiple-source  15
 painting with, defined  6
 painting with, instantiation  7
 painting with, selecting  7
 uncolored  15
PATusp  8, 31, 36
PatWidth  18
printer
 Level 1 device  13
 Level 2 device  13
 unknown  14
proc  24, 26
R
restore  8
S
setcachedevice  29
setcharwidth  29
setcolor  8
setcolorspace  8
setgstate  33
setmatrix  35
setpattern  34
spacing
 inter-cell  16
strokepath  19, 34
 clip  19
T
tilemtx  35
tiling
 snapped pattern space  17
TilingTypes  17, 19, 29, 35
Type 3 font  15
U
undefined error  12
unset pattern. See PATusp
V, W
 veneers
 Level 2  37
 makepattern  13
 PATmp  37
 PATsc  38
 PATsp  38
 PATusp  39
 setpattern  13
VMerror  6
X
XStep  16, 29
Y, Z
YStep  16