POSTSCRIPT

L A N G U A G E

COLOR OPERATOR DEFINITIONS

ADOBE SYSTEMS

I N C O R P O R A T E D

POSTSCRIPT Language Color Operator Definitions

April 25, 1988

1 INTRODUCTION

The POSTSCRIPT language will be upgraded to support color more fully by a variety of means. This document describes how Adobe Systems Incorporated will upgrade the language and how we will provide backward compatibility in existing printers.

The POSTSCRIPT language currently supports color to a limited extent. The **setrgbcolor** and **sethsbcolor** operators allow filled regions, strokes, image masks, and characters to be printed in color. These operators are implemented on black-and-white machines by generating an equivalent gray shade and using that shade when printing. While the set of operators currently supported is small, it is sufficient to produce nearly all synthetic color pictures. In fact, the Adobe Illustrator program supports the painting of objects with colors, and it can generate output files with calls on the existing POSTSCRIPT language color operators.

To support color printing completely, however, a few additional operators need to be added to the POSTSCRIPT language to fill the gaps not covered by **setrgbcolor** and **sethsbcolor**. Most significantly, a **colorimage** operator, similar in operation to the current **image** operator, but working with red-green-blue (RGB) and cyan-magenta-yellow-black (CMYK) color input and generating full-color output, is required. There are also some operators that need to be added to control each of three or four color outputs individually.

The POSTSCRIPT language color model supports one-color, three-color, and four-color output devices. The devices may be binary (one bit per pixel per color component) or gray-scale (multiple bits per pixel per color component, representing a range of intensities of each color component). A binary device uses halftoning to produce intermediate shades of its color components. If a device has eight bits per pixel, it is called a *full gray-scale* device, and uses no halftoning. Devices with more than one and fewer than eight bits per pixel use a combination of built-in intensities and halftoning to produce the full range of desired shades of its color components. Three-color devices may be either red-green-blue (RGB) or cyan-magenta-yellow (CMY). Four-color devices are cyan-magenta-yellow-black (CMYK).

All new operators will have a POSTSCRIPT language implementation for black-and-white machines as well as a built-in implementation for both black-and-white machines and color machines. New implementations of the POSTSCRIPT interpreter will contain an integrated, efficient implementation of the new operators. Older POSTSCRIPT interpreters can be augmented by a POSTSCRIPT language implementation of the new operators.

2 NEW CONCEPTS

CMYK Color Model

A color component can be characterized by either how much light it reflects or how much light it absorbs. Three standard printing process colors, *cyan*, *magenta*, and *yellow*, each absorb one of the standard light components, *red*, *green*, and *blue*, respectively. *Black*, a fourth standard printing process color, absorbs all components of light. In the red-green-blue (RGB) color model, a color is associated with a real number between 0.0 and 1.0, inclusive, for each of its components red, green, and blue, where 0.0 represents dark (no light) and 1.0 represents full light. In the cyan-magenta-yellow-black (CMYK) model, the four components are again real numbers in the range 0.0 to 1.0 inclusive, but in this case 0.0 represents full light (no ink), and 1.0 represents dark (full ink). Since cyan is the absence of red light, we can state that

$$\text{cyan} = 1.0 - \text{red}.$$

Similarly,

$$\text{magenta} = 1.0 - \text{green}$$

and

$$\text{yellow} = 1.0 - \text{blue},$$

so that, for example, a color that is 0.2 red, 0.7 green, and 0.4 blue can also be expressed as $1.0 - 0.2 = 0.8$ cyan, $1.0 - 0.7 = 0.3$ magenta, and $1.0 - 0.4 = 0.6$ yellow. To improve the fidelity of blacks and grays, a fourth process color, black, is often available on color printers. Just as red is the opposite of cyan, a black value has a sense opposite to that of a POSTSCRIPT language gray value; that is,

$$\text{black} = 1.0 - \text{gray}.$$

The CMYK color model uses these cyan, magenta, yellow, and black component values to describe a color.

Black Generation and Undercolor Removal

In converting from the RGB color model to the CMYK color model, a direct subtraction of each color component from 1.0 will only yield values for cyan, magenta, and yellow. We may wish to determine a black value and to adjust the cyan, magenta, and yellow values to compensate for the amount of black added.

For some printing processes, it may be desirable to generate a black value exactly equal to the minimum of the cyan, magenta and yellow values, or it may be desirable to generate no black, or even extra black. The **setblackgeneration** operator (see section 3) allows the user a good deal of flexibility to handle all of these cases. This operator takes as its argument a procedure that maps the minimum of the *user cyan*, *magenta*, and *yellow* color components to *black* values. This mapping is performed whenever a color is set through the use of **setrgbcolor**, the three-color case of **colorimage** (see section 3), or, indirectly, **sethsbcolor**. The black values will then be mapped to *device black* values by applying the gray transfer function to their difference from 1.0 and subtracting the result from 1.0 (see **setcolortransfer** in section 3). For colors set through the use of **setgray**, **setcmykcolor** (see section 3), or the one- or four-color cases of **colorimage**, black generation is not applied. The black value thus computed is only used for output on four-color output devices.

Undercolor removal can be used to compensate for having created a black component through black generation. The **setundercolorremoval** operator (see section 3) can be used to remove exactly the same amount of color from each of the cyan, magenta, and yellow components as was generated to make the black component. Alternatively, the **setundercolorremoval** operator can set up a function that always returns zero, resulting in no color being removed from the cyan, magenta and yellow components, or it can be set up to remove some fraction of the black amount. As is the case with **setblackgeneration**, this operator permits considerable flexibility in color correction. The **setundercolorremoval** operator has as its argument a procedure that maps the minimum of the *user cyan*, *magenta*, and *yellow* color components to a value to be subtracted from each of these same components. This mapping is performed whenever a color is set through the use of **setrgbcolor**, the three-color case of

colorimage (see section 3), or, indirectly, **sethsbcolor**. After subtracting the value generated in the above mapping from the color components, and resetting negative values to 0.0 and values greater than 1.0 to 1.0, each component will be subtracted from 1.0 to yield *red*, *green*, and *blue* components, and each of these components will be mapped into a *device color component* using its respective transfer function (see **setcolortransfer** in section 3). For colors set through the use of **setgray**, **setcmykcolor**, or the one- or four-color cases of **colorimage**, undercolor removal is not applied. Undercolor removal only affects the output from four-color devices.

The following equations define the complete color transformation process. The values red_u, grn_u, blu_u, gry_u, cyn_u, mag_u, yel_u, and blk_u are the (input) values of *user red*, *green*, *blue*, *gray*, *cyan*, *magenta*, *yellow*, and *black*, respectively. *UCR()* is the undercolor removal function. *BG()* is the black generation function. *RedT*, *GrnT*, *BluT*, and *GryT* are the red, green, blue, and gray transfer functions (see **setcolortransfer** in section 3), respectively. The values red_d, grn_d, blu_d, gry_d, cyn_d, mag_d, yel_d, and blk_d are the (output) values of *device red*, *green*, *blue*, *gray*, *cyan*, *magenta*, *yellow*, and *black*, respectively.

$$
\begin{aligned}
&cyn_u = 1.0 - red_u\\
&mag_u = 1.0 - grn_u\\
&yel_u = 1.0 - blu_u\\
&gry_u = 1.0\\
&blk_u = 0.0\\
&k = \mathrm{Min}\{cyn_u, mag_u, yel_u\}\\
&u = \mathrm{UCR}(k)\\
&red_d = \mathrm{RedT}(1.0 - \mathrm{Max}\{0.0, cyn_u - u\})\\
&grn_d = \mathrm{GrnT}(1.0 - \mathrm{Max}\{0.0, mag_u - u\})\\
&blu_d = \mathrm{BluT}(1.0 - \mathrm{Max}\{0.0, yel_u - u\})\\
&gry_d = \mathrm{GryT}(1.0 - \mathrm{BG}(k))\\
&cyn_d = 1.0 - red_d\\
&mag_d = 1.0 - grn_d\\
&yel_d = 1.0 - blu_d\\
&blk_d = 1.0 - gry_d
\end{aligned}
$$

CMYK Color Selection

For the most demanding cases, color matching may require more complicated methods than those described above. The **setcmykcolor** operator and the four-color case of the **colorimage** operator are provided for the user who knows what the resulting cyan, magenta, yellow, and black color components should be on a particular device, and this operator will bypass the black generation and undercolor removal operations. Since this operator provides no correction other than the transfer functions as set using **setcolortransfer** (see section 3), the results will be device dependent.

The following equations define the complete color transformation process for the **setcmykcolor** operator and the four-color case of the **colorimage** operator. The values are as defined in *Black Generation and Undercolor Removal* above.

$$
\begin{aligned}
red_u &= 1.0 - cyn_u \\
grn_u &= 1.0 - mag_u \\
blu_u &= 1.0 - yel_u \\
gry_u &= 1.0 - blk_u \\
red_d &= RedT(red_u) \\
grn_d &= GrnT(grn_u) \\
blu_d &= BluT(blu_u) \\
gry_d &= GryT(gry_u) \\
cyn_d &= 1.0 - red_d \\
mag_d &= 1.0 - grn_d \\
yel_d &= 1.0 - blu_d \\
blk_d &= 1.0 - gry_d
\end{aligned}
$$

Color Screens, Transfer Functions, and Images

The new operators **setcolorscreen**, **currentcolorscreen**, **setcolortransfer**, and **currentcolortransfer** provide a multiple-color expansion of the operators **setscreen**, **currentscreen**, **settransfer**, and **currenttransfer**, respectively, by setting up a screen and a transfer function for each color component. The **colorimage** operator provides an expansion of the **image** operator to allow samples of one, three, or four color components. (See section 3 for more details.)

3 NEW OPERATORS

setcmykcolor

cyan magenta yellow black **setcmykcolor** –

sets the current color parameter in the graphics state to a color described by the parameters *cyan*, *magenta*, *yellow*, and *black*, each of which must be a number in the range 0.0 to 1.0. This establishes the color used subsequently to paint shapes such as lines, areas, and characters on the current page. This operator bypasses the black generation and undercolor removal operations.

For additional information, see section 2.

Note that applying the **setgray** operator sets the gray color component to its single argument value and the red, green, and blue color components to 1.0. It is the equivalent of the following use of **setcmykcolor**:

```
0.0 0.0 0.0 1.0 5 –1 roll sub setcmykcolor
```

The use of **setcmykcolor** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is not permitted (an **undefined** error results).

ERRORS:
stackunderflow, typecheck

currentcmykcolor

– **currentcmykcolor** cyan magenta yellow black

returns the four components of the current color in the graphics state according to the cyan-magenta-yellow-black color model (see **setcmykcolor**).

Note that the **currentgray** operator returns a weighted average of all four color components. Applying it is the equivalent of the following use of **currentcmykcolor**:

```
1.0 currentcmykcolor 4 1 roll 0.11 mul 3 1 roll 0.59 mul
exch 0.30 mul add add add sub dup 0.0 lt {pop 0.0} if
```

ERRORS:
stackoverflow

setcolortransfer redproc greenproc blueproc grayproc **setcolortransfer** –

sets the current transfer function parameters for red, green, blue, and gray in the graphics state. Each operand must be a POSTSCRIPT language procedure that may be called with a number in the range 0.0 to 1.0 (inclusive) on the operand stack and will return a number in the same range. These procedures map user values of the color components (e.g., those specified by **setrgbcolor** and adjusted by **setblackgeneration** and **setundercolorremoval**, or 1.0 minus those specified by **setcmykcolor**) to *device* color components (for halftones, a weighted average of the lightness of pixels in a halftone cell). Only those transfer functions corresponding to color components supported by a device will have an effect on that device's output. For example, *redproc*, *greenproc*, and *blueproc* will have no effect on a black-and-white device, while *grayproc* will have no effect on an RGB device.

The **settransfer** operator takes a single procedure argument whose purpose is to provide gamma correction for a printer's halftoning response. That operator is useful for a variety of effects beyond its original intention as a gray response correction function, but it is useful only in the context of a single output color, as on black-and-white printers. The **setcolortransfer** operator is the logical expansion of **settransfer** to four color components; it takes four function arguments, each similar in purpose to the function argument of **settransfer**, but each function separately controls the response for each of the red (1.0 minus cyan), green (1.0 minus magenta), blue (1.0 minus yellow) and gray (1.0 minus black) outputs, respectively.

Applying the **settransfer** operator sets all four transfer functions equally. It is the equivalent of the following use of **setcolortransfer**:

```
dup dup dup setcolortransfer
```

EXAMPLE:

```
{} {} {dup mul} {} setcolortransfer
```

This sets device blue as the square of user blue and leaves the other color components unchanged.

The use of **setcolortransfer** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is not permitted (an **undefined** error results).

ERRORS:

stackunderflow, typecheck

currentcolortransfer

– **currentcolortransfer** redproc greenproc blueproc grayproc

returns the current transfer functions in the graphics state for each of the four color components (see **setcolortransfer**).

The **currentcolortransfer** operator is the logical expansion of **currenttransfer** to four color components. Applying the **currenttransfer** operator returns the gray transfer function. It is the equivalent of the following use of **currentcolortransfer**:

```
currentcolortransfer 4 1 roll pop pop pop
```

ERRORS:
stackoverflow

setblackgeneration

proc **setblackgeneration** –

sets the current black generation function parameter in the graphics state. The *proc* operand must be a POSTSCRIPT language procedure that may be called with a number in the range 0.0 to 1.0 (inclusive) on the operand stack and will return a number in the same range. This procedure maps the minimum of the *user cyan*, *magenta*, and *yellow* color components to *black* values.

For additional information, see section 2.

EXAMPLE:

```
{dup .75 le {pop 0.0} {.75 sub 4.0 mul} ifelse} setblackgeneration
```

This sets the black component to zero when the minimum of cyan, magenta, and yellow is less than or equal to .75, while minima greater than .75 produce a black component that increases linearly from 0.0 (at a minimum of .75) to 1.0 (when user cyan, magenta, and yellow all have values of 1.0).

The use of **setblackgeneration** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is not permitted (an **undefined** error results).

ERRORS:
stackunderflow, typecheck

currentblackgeneration

– **currentblackgeneration** proc

returns the current black generation function in the graphics state (see **setblackgeneration**).

ERRORS:
stackoverflow

setundercolorremoval

proc **setundercolorremoval** –

sets the current undercolor removal function parameter in the graphics state. The *proc* operand must be a POSTSCRIPT language procedure that may be called with a number in the range 0.0 to 1.0 (inclusive) on the operand stack and will return a number in the range –1.0 (to *increase* the color components) to +1.0 (to *decrease* the color components). This procedure maps the minimum of the *user cyan*, *magenta*, and *yellow* color components to a value to be subtracted from each of these same components.

For additional information, see section 2.

EXAMPLE:

```
{currentblackgeneration exec .5 mul} setundercolorremoval
```

This sets the undercolor removal to be half of the black component from black generation.

The use of **setundercolorremoval** after a **setcachedevice** operation within the scope of a **BuildChar** procedure is not permitted (an **undefined** error results).

ERRORS:
stackunderflow, typecheck

currentundercolorremoval

– **currentundercolorremoval** proc

returns the current undercolor removal function in the graphics state (see **setundercolorremoval**).

ERRORS:
stackoverflow

setcolorscreen redfrequency redangle redproc greenfrequency greenangle greenproc bluefrequency blueangle blueproc grayfrequency grayangle grayproc **setcolorscreen** –

sets the current halftone screen definitions for red, green, blue, and gray in the graphics state. Each of the *redfrequency*, *greenfrequency*, *bluefrequency*, and *grayfrequency* operands is a number that specifies the screen frequency for one color component, measured in halftone cells per inch in device space. The *redangle*, *greenangle*, *blueangle*, and *grayangle* operands specify the number of degrees by which their respective halftone screens are to be rotated with respect to the device coordinate system. The *redproc*, *greenproc*, *blueproc*, and *grayproc* operands are each a POSTSCRIPT language procedure defining one color component's spot function, which determines the order in which pixels within a halftone cell of that color component are assigned a light value to produce any desired shade of that color component. The red, green, and blue screens will have no effect on a black-and-white device, the gray screen will have no effect on an RGB or CMY device, and no screens will have any effect on a full (8 bits per pixel) gray-scale device.

Many binary color printers (those that print each dot at full intensity or not at all) require different angles for each color-component halftone in order to have attractive output. Each color printer containing a POSTSCRIPT interpreter will have a default color screen chosen to look good on that printer.

The **setcolorscreen** operator is the logical expansion of **setscreen** to four color components. It takes the same three types of arguments as **setscreen**, but repeated four times. Applying the **setscreen** operator sets all four screens equally. It is the equivalent of the following use of **setcolorscreen**:

```
3 copy 6 copy setcolorscreen
```

EXAMPLE:

```
/sfreq 50 def                % 50 halftone cells per inch
/sproc {dup mul exch dup mul add 1 exch sub} def
                             % dot-screen spot function
sfreq 75 /sproc load         % 75 degree red (cyan) screen
sfreq 15 /sproc load         % 15 degree green (magenta) screen
sfreq 0 /sproc load          % 0 degree blue (yellow) screen
sfreq 45 /sproc load         % 45 degree gray (black) screen
setcolorscreen
```

ERRORS:

limitcheck, rangecheck, stackunderflow, typecheck

currentcolorscreen

– **currentcolorscreen** redfrequency redangle redproc greenfrequency greenangle greenproc bluefrequency blueangle blueproc grayfrequency grayangle grayproc

returns all twelve current halftone screen parameters in the graphics state (see **setcolorscreen**).

The **currentcolorscreen** operator is the logical expansion of **currentscreen** to four color components. Applying the **currentscreen** operator returns the three parameters describing the gray screen. It is the equivalent of the following use of **currentcolorscreen**:

```
currentcolorscreen 12 3 roll 9 {pop} repeat
```

ERRORS:
stackoverflow

colorimage width height bits/sample matrix $proc_0$ [... $proc_{ncolors-1}$] multiproc ncolors **colorimage** –

renders a sampled image in 1, 3, or 4 color components onto the current page. The first four arguments are the same as those for the **image** operator. The *bits/sample* argument applies equally to all color components. The **colorimage** operator permits its $proc_i$ arguments to return RGB or CMYK sample values rather than single color (gray) values as are returned by the **image** operator's *proc* argument. The *ncolors* argument, a 1, 3, or 4, is the number of color components represented in the samples. If this argument is 1, the image only has gray samples and is equivalent to an image described by the **image** operator with the same initial five arguments. If the *ncolors* argument is 3 or 4, the **colorimage** operator will take RGB (light-high) or CMYK (dark-high) samples, respectively. On a four-color device, a CMYK color image will bypass the black generation and undercolor removal operations, whereas an RGB color image will include these operations. The *multiproc* argument is a boolean that distinguishes between two forms of the **colorimage** operator: the *false* form uses one procedure argument ($proc_0$), whereas the *true* form uses one procedure argument per sample color ($proc_0$ ... $proc_{ncolors-1}$); i.e., three procedure arguments for RGB or four procedure arguments for CMYK. If the *ncolors* argument is 1, then there is only one procedure argument, $proc_0$, regardless of the value of the *multiproc* argument. For a detailed description of the data formats and how the $proc_i$ procedures are called, see *Data formats for colorimage operator* below.

The **colorimage** operator is the logical expansion of **image** to multiple color components. Applying the **image** operator renders a one-color image, equivalent to the following uses of **colorimage**:

```
false 1 colorimage
```

or

```
true 1 colorimage
```

Use of **setcolorscreen**, **setcolortransfer**, **setscreen**, or **settransfer** by any of the $proc_i$ procedures will cause unpredictable results. Use of the **colorimage** operator after a **setcachedevice** within the context of a **BuildChar** procedure is not permitted (an **undefined** error results).

ERRORS:
limitcheck, rangecheck, stackunderflow, typecheck, undefined, undefinedresult

Data formats for colorimage operator

As stated above, the **colorimage** operator has two forms, distinguished by its *multiproc* argument.

The single-procedure form is most useful if sample input is taken from a source that has already merged the color components. This form provides samples for which each RGB triple or CMYK quadruple is packed together in the string result of the procedure, using one of the following bit formats (where the high-order bit is shown on the left):

Bits/ sample	**RGB Format**				
1	RGBRGBRG	BRGBRGBR	GBRGBRGB	RGBRGBRG	....
2	RRGGBBRR	GGBBRRGG	BBRRGGBB	RRGGBBRR	....
4	RRRRGGGG	BBBBRRRR	GGGGBBBB	RRRRGGGG	....
8	RRRRRRRR	GGGGGGGG	BBBBBBBB	RRRRRRRR	....

or

Bits/ sample	**CMYK Format**				
1	CMYKCMYK	CMYKCMYK	CMYKCMYK	CMYKCMYK	....
2	CCMMYYKK	CCMMYYKK	CCMMYYKK	CCMMYYKK	....
4	CCCCMMMM	YYYYKKKK	CCCCMMMM	YYYYKKKK	....
8	CCCCCCCC	MMMMMMMM	YYYYYYYY	KKKKKKKK	....

The multiple-procedure form uses each procedure to return a string of sample values for only one color component, using the same format as strings returned by the *proc* argument of the **image** operator. For a three-color image, $proc_0$ returns red values, $proc_1$ returns green values, and $proc_2$ returns blue values. Likewise, for a four-color image, $proc_0$ returns cyan values, $proc_1$ returns magenta values, $proc_2$ returns yellow values, and $proc_3$ returns black values. The **colorimage** operator will call each of these procedures in turn, starting with $proc_0$ and continuing with $proc_1$, $proc_2$, and, if available, $proc_3$. the red or cyan procedure and continuing with green and blue or magenta, yellow, and black. When the *colorimage* operator needs more samples, it will call these procedures again in the same order. *The color procedures must use separate strings for the three or four results of the three or four procedures;* e.g., reusing the red string for the green values may result in some of the red values being lost. Also, *the three or four procedures must return strings of identical lengths within each cycle of three or four calls, respectively*.

The multiple-procedure form is most useful when color sample data are taken from separate color scanner passes. The **colorimage** operator requires the color data to be interleaved, as it must have all three or four components of any sample at the same time to do its work. The single-procedure form interleaves the data at the sample level; this may be convenient only if the data are already in that form when preparing the POSTSCRIPT language page description. The multiple-procedure form allows the interleaving to be performed at a much coarser level. Typically, each procedure of the multiple-procedure form will return some number of scan lines of samples, where the number of samples returned at each call is limited by the string storage available in the POSTSCRIPT interpreter.

Examples for colorimage operator

The following examples illustrate the use of the **colorimage** operator:

EXAMPLE 1:

```
/rgbstr 192 string def     % string to hold 256 two-bit samples
                           %   each of red, green, and blue data
45 140 translate           % locate lower left corner of image
132 132 scale              % map image to 132 point square
256 256 2                  % dimensions of source image
[256 0 0 –256 0 256]       % map unit square to source
{currentfile               % read image data from program file
  rgbstr readhexstring pop}
false 3                    % single proc, 3 colors, bit format:
                           %   rrggbbrr ggbbrrgg bbrrggbb ...
colorimage
94a1bec8c0b371a3a5c4d281 ...      (98304 hex digits of image data)
```

The above is a one-procedure 2-bit RGB image. The base-4 representation of the hex data is

```
2110 2201 2332 ...
```

which is composed of the following color samples:

```
r=2 g=1 b=1 r=0   g=2 b=2 r=0 g=1   b=2 r=3 g=3 b=2 ...
```

EXAMPLE 2:

```
/rstr 256 string def        % string to hold 256 8-bit red samples
/gstr 256 string def        % string to hold 256 8-bit green samples
                            %   (distinct from rstr)
/bstr 256 string def        % string to hold 256 8-bit blue samples
                            %   (distinct from rstr and bstr)
45 140 translate            % locate lower left corner of image
132 132 scale               % map image to 132 point square
256 256 8                   % dimensions of source image
[256 0 0 –256 0 256]        % map unit square to source
{currentfile rstr readhexstring pop}
                            % read red data from program file
{currentfile gstr readhexstring pop}
                            % read green data from program file
{currentfile bstr readhexstring pop}
                            % read blue data from program file
true 3                      % multiple proc, 3 colors
colorimage
7b5e606969615365556a6a66 ...      (512 hex digits of red data)
88868d848a92878578787a82 ...      (512 hex digits of green data)
62717c7b736e707d7b6a7c79 ...      (512 hex digits of blue data)
7d8b8d8c837d8b8e9284878e ...      (512 hex digits of red data)
2788b838b8e8e86868988908 ...      (512 hex digits of green data)
81817d857f85858290949487 ...      (512 hex digits of blue data)
... (390144 more hex digits of RGB data, cycling as above)
```

The above is a three-procedure 8-bit RGB image. The initial samples for each color, in hex, are

red:	7b	5e	60	69	...
green:	88	86	8d	84	...
blue:	62	71	7c	7b	...

EXAMPLE 3:

```
/cstr 128 string def        % string to hold 256 4-bit cyan samples
/mstr 128 string def        % string to hold 256 4-bit magenta samples
                            %   (distinct from cstr)
/ystr 128 string def        % string to hold 256 4-bit yellow samples
                            %   (distinct from cstr and mstr)
/kstr 128 string def        % string to hold 256 4-bit black samples
                            %   (distinct from cstr, mstr, and ystr)
45 140 translate            % locate lower left corner of image
132 132 scale               % map image to 132 point square
256 256 4                   % dimensions of source image
[256 0 0 –256 0 256]        % map unit square to source
{currentfile cstr readhexstring pop}
                            % read cyan data from program file
{currentfile mstr readhexstring pop}
                            % read magenta data from program file
{currentfile ystr readhexstring pop}
                            % read yellow data from program file
{currentfile kstr readhexstring pop}
                            % read black data from program file
true 4                      % multiple proc, 4 colors
colorimage
e1d8caa57b655b6779606b72 ...     (256 hex digits of cyan data)
6bdbb867b9fb6a4859569989 ...     (256 hex digits of magenta data)
996796e639cc0b29f94736c7 ...     (256 hex digits of yellow data)
c9c0cad0d3cad2b7c9e2d7d8 ...     (256 hex digits of black data)
5d2d6d7d4d3d1d4d6c9d4d9d ...     (256 hex digits of cyan data)
4cdcfd4d6d1d8d7d5d4d2d2d ...     (256 hex digits of magenta data)
d2b7c9e2d7d8d8cbbac2d9d8 ...     (256 hex digits of yellow data)
88ae96632a70f6f4d8d9d9d8 ...     (256 hex digits of black data)
... (260096 more hex digits of CMYK data, cycling as above)
```

The above is a four-procedure four-bit CMYK image. The initial samples for each color, in hex, are

```
cyan:      e 1 d 8 c a a 5 ...
magenta:   6 b d b b 8 6 7 ...
yellow:    9 9 6 7 9 6 e 6 ...
black:     c 9 c 0 c a d 0 ...
```

EXAMPLE 4:

```
/cstr 1024 string def       % string to hold 1024 8-bit cyan samples
/mstr 1024 string def       % string to hold 1024 8-bit magenta samples
                            %   (distinct from cstr)
/ystr 1024 string def       % string to hold 1024 8-bit yellow samples
                            %   (distinct from cstr and mstr)
/kstr 1024 string def       % string to hold 1024 8-bit black samples
                            %   (distinct from cstr, mstr, and ystr)
/cfile (img/smp.c) (r) file def     % binary file containing 1048576 8-bit
                                    %   cyan samples
/mfile (img/smp.m) (r) file def     % binary file containing 1048576 8-bit
                                    %   magenta samples
/yfile (img/smp.y) (r) file def     % binary file containing 1048576 8-bit
                                    %   yellow samples
/kfile (img/smp.k) (r) file def     % binary file containing 1048576 8-bit
                                    %   black samples
36 126 translate            % locate lower left corner of image
540 540 scale               % map image to 540 point square
1024 1024 8                 % dimensions of source image
[1024 0 0 –1024 0 1024]             % map unit square to source
{cfile cstr readstring pop}         % read cyan data from img/smp.c
{mfile mstr readstring pop}         % read magenta data from img/smp.m
{yfile ystr readstring pop}         % read yellow data from img/smp.y
{kfile kstr readstring pop}         % read black data from img/smp.k
true 4                      % multiple proc, 4 colors
colorimage
```

The above is a four-procedure 8-bit CMYK image, with cyan, magenta, yellow, and black samples taken from the files *img/smp.c*, *img/smp.m*, *img/smp.y*, and *img/smp.k*, respectively. This example only applies to a POSTSCRIPT interpreter that has a file system.

4 COLOR IMPLEMENTATIONS

Each POSTSCRIPT interpreter will use default page setups corresponding to the printer to which it is attached. If the attached printer is a direct color, binary device, the standard page setup will produce three- or four-color output. If the attached printer is a gray-scale color device, then the POSTSCRIPT interpreter will use a gray-scale three- or four-color method. If the attached printer is a black-and-white device, then the default page will be a single black-and-white rendition of the page described.

Black-and-white printers will also have the capability of producing color separations. These are three or four black-and-white pages for each page described. Each of the output pages corresponds to the output for one color component. These separations are only useful if they are then used in another printing process in which they are combined with color to form the intended full-color output. The method by which a POSTSCRIPT language program informs the POSTSCRIPT interpreter that color separations are desired is still to be determined. This color separation capability will not be provided in the backward compatibility package, as it requires considerable internal changes in the POSTSCRIPT interpreter.

Index

Contents