SANFACE Software

data2pdf

Project

A data2pdf based application

Author: **SANFACE Software**
Number of pages: **3**
Created: **27 march 1999**
Modified: **17 may 1999**
Revision: **1**

Index

1. Introduction

SANFACE Software enveloped a new tool **data2pdf** , based on txt2pdf core.
The goal of **data2pdf** is very simple:
give to the user one or more white PDF pages, where with a language and the PDF syntax the user can design and write what he wants.
From version 1.5 the user can import also JPEG image.

2. Syntax

The **data2pdf** language is simple and power, the approach is very similar to HMTL (from <tag> … </tag> to #!tag# … #!/tag#). To permit a major versatility is possible to use directly the PDF syntax to design and write.
We can divide the operators in unique operators, line operators, multilines operators, mix operators.
We can also divide the operator in document operators, background operators, page operators.

2.1. Unique operators

A unique operator needs to start ad the begin of a new line and end to the end of the line. The landscape operator is an example:

```
#!landscape#
```

It doesn't require an analogue close operator.

2.2. Line operators

The line operator syntax is #!tag# … #!/tag#
A line operator needs to start ad the begin of a new line and end to the end of the line. The paper operator is an example:

```
#!paper#a4#!/paper#
```

2.3. Multilines operators

In a multilines operator the open and the close tags are like unique operators.
The background design operator is an example:

```
#!bgdesign#
```

```
.9 g
0 G
5 w
25 25 792 545 re
B
#!/bgdesign#
```

2.4. Mix operators

The line operator syntax is #!tag# … #!/tag#

It can be multilines and the open and close tags can be everywhere inside the text. The bold operator is an example:

```
This is a normal text. #!b#This is a bold text.#!/b# Now it's normal again.
```

2.5. Document operators

A document operator starts at the beginning of the configuration file and permits to set a value for all the PDF document.

An example is the paper operator (see 2.2)

2.6. Background operators

At the moment the background operators are three: background design, background text and background image. Background design permits to define a background design that will be used in every PDF page. The same for background text. They are multiline operators.

Background image is a linear operator.

2.7. Page operators

The validity of page operators is only inside the current page description.

An example is the design operator:

```
#!design#
…
#!/design#
```

3. Structure

It's important to understand the sequence of the operator to construct correctly the configuration file:

document operators

background operators

page operator

…

page operator

Inside page operator there is the page substructure:

image

design

text

Inside text operator there is the text substructure:

textcommand

text with the possibility to mark bold, italic or bolditalic

4. List of the operators

#!font#…#!/font#

linear and document operator. The fonts are: **Courier**, **Helvetica**, **Times**

(IMPORTANT: you can use different fonts or write the same fonts in a different mode. If you wrong to write it the program will use the default font Courier)

#!paper#…#!/paper#

linear and document operator. The paper format are: **letter**, **a3**, **a4**, **a5**, width**x**height (If wrong to write it the program will use the default paper letter)

#!landscape#

unique and document operator

#!title#…#!/title#

#!author#…#!/author#

#!creator#…#!/creator#

#!keywords#…#!/keywords#

#!subject#…#!/subject#

linear and document operator for the INFO fields inside PDF (see the attach from PDF manual)

#!bgdesign#

…

#!/bgdesign#

#!bgtext#

…

#!/bgtext#

multilinear and background operators. Inside you can use the PDF syntax to design and write (see the attach from PDF manual)

#!bgimage#image.jpg;width;height;a_cm;b_cm;c_cm;d_cm;e_cm;f_cm#!/bgimage#

a b c d e f **cm** Modifies the CTM by concatenating the specified matrix. Although the operandsspecify a matrix, they are passed as six numbers, not as an array. (concat) (PDF 1.3 Manual 8: Page Descriptions 323)

#!page#

#!image#image.jpg;width;height;a_cm;b_cm;c_cm;d_cm;e_cm;f_cm#!/image#

#!design#

…

#!/design#

#!text#

…
#!/text#
#!/page#
This is the structure of a page: inside page you can use image, design then text.
Inside design you can use the PDF syntax (see the attach from PDF manual).
Inside text you can write normal text with these defaults:
font: the selected font with font size = 10
color font: black
start text point: 50 , page height - 40 (the page height depends by the selected paper)
vertical tab (the distance from two line): 12
If you want to change the defaults you can use
#!textcommand#…#!/textcommand#
with the PDF syntax and
#!fontsize#…#!/fontsize#
to change the default font size.
Inside the the test you can use:
#!b#…#!/b# **bold**
#!i#…#!/i# *italic*
#!bi#…#!/bi# ***bold italic***
Two new operators are:
#!link#link;rect_x1;rect_y1;rect_x2;rect_y2#!/link#
#!circle#x;y;r#!/circle# (you can use circle option only inside design part)

5. Fonts

After the selection of the font with the option #!font#…#!/font#
/F1 is the normal
/F2 is the italic
/F3 is the bold
/F4 is the bold-italic
From version 1.5 now you can use
/F5 is Symbol
/F6 is ZapfDingbats
It's also supported WinAnsiEncoding

6. An example

```
#!font#Courier#!/font#
#!paper#a4#!/paper#
#!landscape#
#!title#Project#!/title#
```

```
#!author#SANFACE Software#!/author#
#!creator#data2pdf#!/creator#
#!keywords#Sanface#!/keywords#
#!subject#Project by SANFACE Software#!/subject#
#!bgdesign#
.9 g
0 G
5 w
25 25 792 545 re
B
#!/bgdesign#
#!bgtext#
/F1 15 Tf
0 0 0 rg
1 0 0 1 650 40 Tm
(SANFACE Software) Tj
#!/bgtext#
#!page#
#!design#
3 w
.5 g
200 250 400 70 re
B
#!/design#
#!text#
#!textcommand#1 0 0 1 240 285 Tm#!/textcommand#
#!fontsize#40#!/fontsize#
Project
#!/text#
#!/page#
#!page#
#!design#
1 w
.7 g
27 75 788 50 re
B
27 175 788 50 re
B
27 275 788 50 re
B
27 375 788 50 re
B
27 475 788 50 re
B
#!/design#
#!text#
#!textcommand#50 TL#!/textcommand#
#!textcommand#1 0 0 1 50 545 Tm#!/textcommand#
ISOLatin1Encoding Test
Hyötyläinen
Tab Test
#!fontsize#15#!/fontsize#
```

```
qqqqqqq    qqqqqq    qqqqq     qqqq qqq  qq   q
p     pp   ppp  pppp ppppp     pppppp    ppppppp
#!textcommand#0 0 1 rg#!/textcommand#
Style test
#!b#bold#!/b#      normal      #!i#italic#!/i#      normal
#!bi#bolditalic#!/bi#
#!/text#
#!/page#
```

In the document part we have selected:

Courier font

A4 paper format (595,842) in landscape mode (842,595)

We have set the author, title, creator, keywords, subject INFO PDF fields.

The background design is a rectangle (re command) with the border black (0 G), large 5 (5 w) and inside gray (.9 g).

The background test (SANFACE Software) is black (0 0 0 rg), with font size 15 and start at 650, 40 (0,0 is the right down angle of the page).

The first page has a rectangle with the border line black (it uses the precedent G definition), large 3 and inside gray (.5 g).

The text (Project) has e font size 40 and start at point 240,285.

The second page has 5 rectangles with the border line black, large 1 and inside gray (.3 g). The distance between them is 100.

The vertical tab of the test is set to 50 and start at 50,545

Then we change the font size to 15 and the color of the text to blue (0 0 1 rg).

We mark bold, italic and bolditalic a few word.

CHAPTER 8

Page Descriptions

This chapter describes the PDF operators that draw text, graphics, and images on the page and in other "marking contexts" such as forms and patterns. It completes the specification of PDF. The following chapters describe how to produce efficient PDF files.

Text, graphics, and images are drawn using the coordinate systems described in Chapter 3. It may be useful to refer to that chapter when reading the description of various operators, to obtain a better understanding of the coordinate systems used in PDF documents and the relationships among them.

Appendix B contains a complete list of operators, arranged alphabetically.

Note *Throughout this chapter, PDF operators are shown with a list of the operands they require. For operators that correspond to one or more PostScript language operators, the corresponding PostScript language operator appears in bold on the first line of the operator's definition. An operand specified as "number" may be either an integer or a real number. Otherwise, numeric operands must be integers.*

8.1 Overview

A PDF page description can be considered a sequence of graphics objects. These objects generate marks that are applied to the current page, obscuring any existing marks they may overlay.

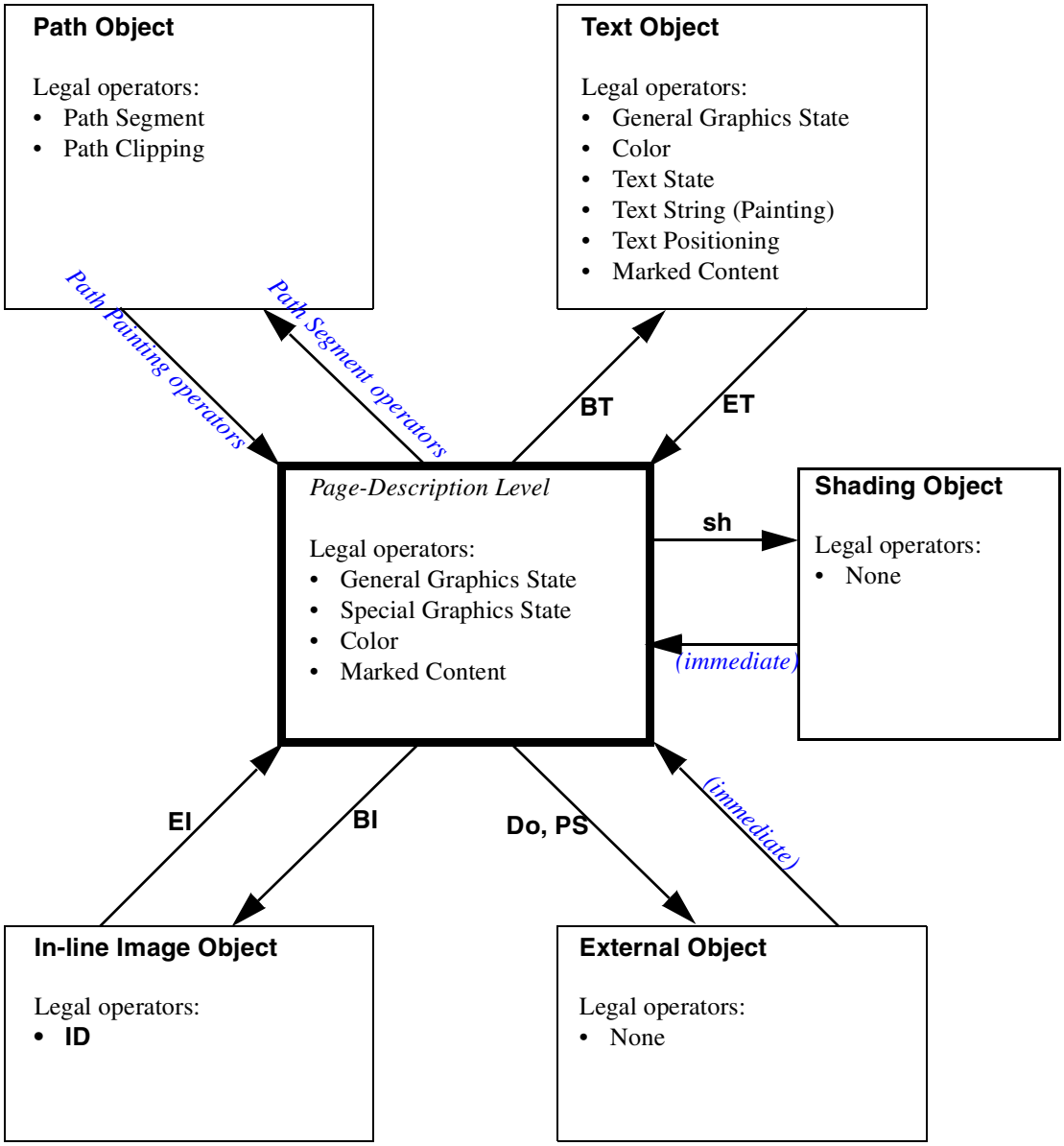
PDF provides five types of graphics objects:

- A *path object* is an arbitrary shape made of straight lines, rectangles, and cubic curves. A path may intersect itself and may have disconnected sections and holes. A path object usually includes a painting operator that specifies whether the path is filled, stroked, and/or serves as a clipping path. A painting operator is not required, however; unpainted ("invisible") path objects are sometimes used as placeholders or to denote text bounds.

- A *text object* consists of one or more character strings that can be placed anywhere on the page and in any orientation. Like a path, text can be stroked, filled, and/or serve as a clipping path.

- An *image object* consists of a set of samples using a specified color model. Images can be placed anywhere on a page and in any orientation.

Figure 8.1 *Graphics Objects*



General Graphics State operators: **d, gs, i, j, J, M, ri, w**
Special Graphics State operators: **q, Q, cm**
Color operators: **g, G, k, K, rg, RG, sc, SC, scn, SCN, cs, CS**
Text State operators: **TC, Tf, TL, Tr, Ts, Tw, Tz**
Text String (Painting) operators: **Tj, TJ, ', "**
Text Positioning operators: **Td, TD, Tm, T***
Path Segment operators: **c, h, l, m, re, v, y**
Path Painting operators: **f, F, f*, n, s, S, b, b*, B, B***
Path Clipping operators: **W, W***
Marked Content operators: **BMC, BDC, EMC, MP, DP**
Shading operator: **sh**

- An *External Object* (XObject) is an object defined outside the stream. The interpretation of an XObject depends on its type. PDF currently supports three types of XObjects: images, forms, and PostScript language fragments.

- A*shading object* describes a smoth transition of colors across an area on the page.

As described in Chapter 7, a PDF page description is not necessarily self-contained. It often contains references to resources such as fonts, patterns, forms, or images not found within the page description itself but located elsewhere in the PDF file.

Figure 8.1 shows the ordering rules for the operations that define graphics objects. Some operations are permitted only in certain graphics objects or in the intervals between graphics objects, which is called the Page Description Level in the Figure. Every contents stream begins at the Page Description Level, where changes can be made to the graphics state, including colors and text-specific parameters, as explained in the following sections. The arrows indicate the operators that mark the beginning or end of each of the graphics objects. For example, any Path Segment operator such as **m** (moveto) marks the beginning of a Path object. Inside a Path object, additional Path Segment operators are permitted, as are Path Clipping operators, but not a General Graphics State operator such as **d** (setdash), for example. A Path Painting operator such as **f** (fill) marks the end of the Path Object and the return to the Page Description Level.

8.2 Graphics state

The exact effect of drawing a graphics object is determined by parameters such as the current line thickness, font, and leading. These parameters are part of the *graphics state*.

Although the contents of the PDF graphics state are similar to those of the graphics state in the PostScript language, there are several differences:

1. In PDF, the graphics state is divided into four distinct groups of parameters and operators. There are specific groups for text, for color, for “generic” marking operations, and for the graphics state itself. In this chapter, starting in Figure 8.1, these are referred to as Text State, Color, General Graphics State, and Special Graphics State operators, respectively. The Text State, for example, enables the implementation of a more compact set of text operators.

2. The graphics state is extended to distinguish the parameters for fill operations from those for stroke operations. The use of separate fill and stroke colors in PDF is necessary to implement painting operators that both fill and stroke a path or text.

3. Finally, the graphics state in PDF 1.2 permits user extensions by means of the Marked Content operators. These have no effect on viewing or printing, but they preserve information that may be of use to plug-ins. **PDF 1.2**

The graphics state is initialized at the beginning of each page, using the default values specified in each of the operator descriptions.

8.3 Special Graphics State

The Special Graphics State refers to parameters that apply to all four types of graphics objects: path, text, image, and external.

PDF provides a *graphics state stack* for saving and restoring the entire graphics state: the General Graphics State, the Color, and the Text State. PDF provides an operator, **q**, that saves a copy of the graphics state onto the graphics state stack. Another operator, **Q**, removes the most recently saved graphics state from the stack and makes it the current graphics state.

8.3.1 Special Graphics state parameters

8.3.1.1 Clipping path

The clipping path restricts the region to which paint can be applied on a page. Marks outside the region bounded by the clipping path are not painted. Clipping paths may be specified either by a path, or by using one of the clipping modes for text rendering. These are described in Section 8.6.3, "Path clipping operators," and Section 8.7.1.7, "Text rendering mode."

8.3.1.2 Current transformation matrix

The CTM is the matrix specifying the transformation from user space to device space. It is described in Section 3.2, "User space."

8.3.1.3 Current point

All drawing on a page makes use of the *current point*. In an analogy to drawing on paper, the current point can be thought of as the location of the pen used for drawing.

The current point must be set before graphics can be drawn on a page. Several of the operators discussed in Section 8.6.1, "Path segment operators," set the current point. As a path object is constructed, the current point is updated in the same way as a pen moves when drawing graphics on a piece of paper. After the path is painted using the operators described in Section 8.6.2, "Path painting operators," the current point is undefined.

The current point also determines where text is drawn. Each time a text object begins, the current point is set to the origin of the page's coordinate system. Several of the operators described in Section 8.7.3, "Text positioning operators," change the current point. The current point is also updated as text is drawn using the operators described in Section 8.7.5, "Text string operators".

8.3.2 Special Graphics State operators

The operators in this section may be used only at the Page-Description Level; see Figure 8.1. Adjacent to each PDF operator name is the PostScript language equivalent operator, if any.

Arguments	*Operator*	*Semantics*
	q	Saves the current graphics state on the graphics state stack. (**gsave**)
	Q	Restores the graphics state to the most recently saved state. Removes the most recently saved state from the stack and makes it the current state. (**grestore**)
a b c d e f	**cm**	Modifies the CTM by concatenating the specified matrix. Although the operands specify a matrix, they are passed as six numbers, not as an array. (**concat**)

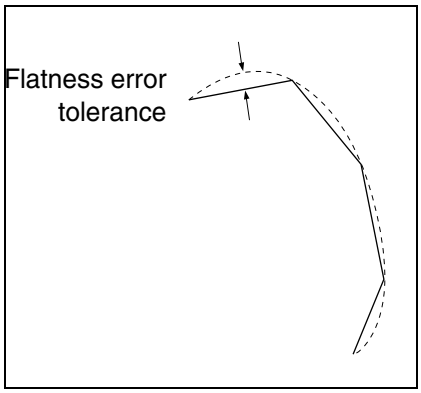
8.4 General Graphics state

8.4.1 Flatness

Flatness sets the maximum permitted distance in device pixels between the mathematically correct path and an approximation constructed from straight line segments, as shown in Figure 8.2.

Note *Flatness is inherently device-dependent, because it is measured in device pixels.*

Figure 8.2 *Flatness*

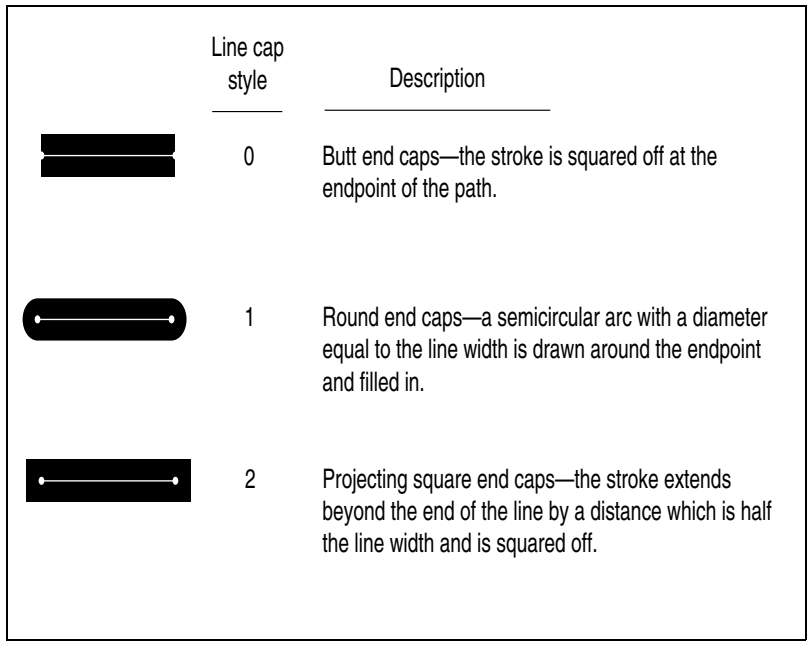


Arguments	*Operator*	*Semantics*
flatness	**i**	Sets the flatness parameter in the graphics state. *flatness* is a number in the range 0 to 100, inclusive. The default value for *flatness* is 0, which means that the device's default flatness is used. (**setflat**)

8.4.2 Line cap style

The line cap style specifies the shape to be used at the ends of open subpaths when they are stroked. Allowed values are shown in Figure 8.3.

Figure 8.3 *Line cap styles*

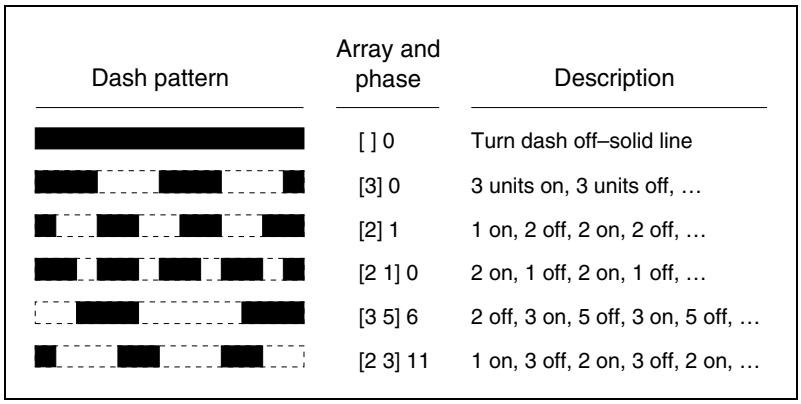


Arguments	*Operator*	*Semantics*
linecap	**J**	Sets the line cap parameter in the graphics state. *linecap* has a default value of 0.

8.4.3 Line dash pattern

The line dash pattern controls the pattern of dashes and gaps used to stroke paths. It is specified by an *array* and a *phase*. The array specifies the length of alternating dashes and gaps. The phase specifies the distance into the dash pattern to start the dash. Both the elements of the array and the phase are measured in user space units. Before beginning to stroke a path, the array is cycled through, adding up the lengths of dashes and gaps. When the sum of dashes and gaps equals the value specified by the phase, stroking of the path begins, using the array from the point that has been reached. Figure 8.4 shows examples of line dash patterns. As can be seen from the figure, the command `[] 0 d` can be used to restore the dash pattern to a solid line. (**setlinecap**)

Figure 8.4 *Line dash pattern*



Dashed lines wrap around curves and corners just as solid stroked lines do. The ends of each dash are treated with the current line cap style, and corners within dashes are treated with the current line join style.

Arguments	*Operator*	*Semantics*
[array] phase	**d**	Sets the dash pattern parameter in the graphics state. If *array* is empty, the dash pattern is a solid, unbroken line; otherwise *array* is an array of numbers, all non-negative and at least one non-zero, that specify alternating distances in user space for the length of dashes and gaps. *phase* is a number that specifies a distance in user space into the dash pattern at which to begin marking the path. The default dash pattern is a solid line. (**setdash**)

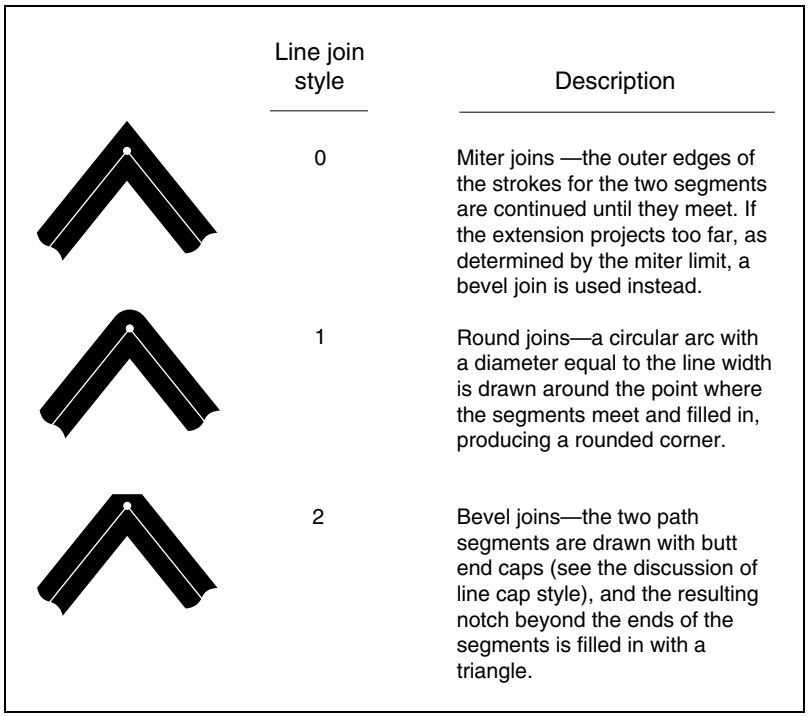
The **S** (stroke) operator uses the contents of the array of dashes and gaps in a cyclical fashion; when it reaches the end of the array, it starts again at the beginning.

When a path consisting of several subpaths is stroked, each subpath is treated independently—in other words, the dash pattern is restarted and *phase* is applied to it at the beginning of each subpath.

8.4.4 Line join style

The line join style specifies the shape to be used at the corners of paths that are stroked. Figure 8.5 shows the allowed values.

Figure 8.5 *Line join styles*



Arguments	*Operator*	*Semantics*
linejoin	**j**	Sets the line join parameter in the graphics state. *linejoin* has a default value of 0. (**setlinejoin**)

8.4.5 Line width

The line width specifies the thickness of the line used to stroke a path and is measured in user space units. A line width of 0 specifies the thinnest line that can be rendered on the output device.

Note *A line width of 0 is an inherently device-dependent value. Its use is discouraged because the line may be nearly invisible when printing on high-resolution devices.*

Arguments	*Operator*	*Semantics*
linewidth	**w**	Sets the line width parameter in the graphics state. *linewidth* is a number and has a default value of 1. (**setlinewidth**)

8.4.6 Miter limit

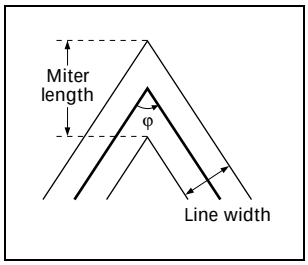
When two line segments meet at a sharp angle and mitered joins have been specified as the line join style, it is possible for the miter to extend far beyond the thickness of the line stroking the path. The miter limit imposes a maximum on the ratio of the miter length to the line width, as shown in Figure 8.6. When the limit is exceeded, the join is converted from a miter to a bevel.

The ratio of miter length to line width is directly related to the angle φ between the segments in user space by the formula:

$$\frac{\text{miter length}}{\text{line length}} = \frac{1}{\sin\left(\frac{\varphi}{2}\right)}$$

For example, a miter limit of 1.415 converts miters to bevels for φ less than 90 degrees, a limit of 2.0 converts miters to bevels for φ less than 60 degrees, and a limit of 10.0 converts miters to bevels for φ less than 11 degrees.

Figure 8.6 *Miter length*



Arguments	*Operator*	*Semantics*
miterlimit	**M**	Sets the miter limit parameter in the graphics state. *miterlimit* is a number that must be greater than or equal to 1, and has a default value of 10. (**setmiterlimit**)

8.4.7 Generic Graphics State operator

All the remaining parameters in the General Graphics State are set with the **gs** operator, whose operand is an "extended graphics state" dictionary. (See page 272.) Each parameter uses a different keyword in this dictionary.

Note *It is expected that any future extensions to the graphics state will also use the* **gs** *operator, with new keywords, rather than new operators.*

Arguments	*Operator*	*Semantics*	
name	**gs**	Sets the specified device-dependent parameters in the graphics state: stroke adjustment, overprint, black generation, undercolor removal, transfer function, halftone, and halftone phase. Parameters that are not specified are not changed. *name* is the name of an ExtGState dictionary in the current Resources dictionary.	**PDF 1.2**

In PDF 1.3, *all* the parameters of the General Graphics State, which can be set by operators described in Sections 8.4.1 through 8.4.6, have equivalent keys in the ExtGState dictionary. This is intended for use by Type 2 patterns (smooth shading), which do not have a contents stream. **PDF 1.3**

8.4.8 Stroke adjustment **PDF 1.2**

The stroke adjustment parameter controls whether the line width and the coordinates of a stroke are automatically adjusted as necessary to produce lines of uniform thickness. For details, see section 7.5.2, "Automatic Stroke Adjustment," of the *PostScript Language Reference Manual, Third Edition* [1]. The keyword for stroke adjustment is **SA**. The default value is *true*.

8.4.9 Overprint **PDF 1.2**

The overprint parameter is used only when producing separations. It specifies whether painting on one separation causes the corresponding areas of other separations to be erased (*false*) or left unchanged (*true*). See section 4.8.4, "Special Color Spaces," of the *PostScript Language Reference Manual, Second Edition* [1].

Separate keywords are used for controlling overprint for fill and for stroke. The keyword **OP** specifies overprint for stroke; the keyword **op** specifies overprint for fill. In an ExtGState dictionary, if **OP** is specified but **op** is omitted, then the **OP** key applies to both fill and stroke. The default value is *false* for both **OP** and **op**. **PDF 1.3**

The dictionary may also specify an *overprint mode* with the **OPM** key. The overprint mode affects the interpretation of the **k** and **K** operators, which set the color in a 4-component CMYK color space (**k** for fill, **K** for stroke). If the overprint mode is 0 (the default), then a color component of 0 erases ("paints white") the current path on the corresponding separation. If the overprint mode is 1, then a color component of 0 leaves the corresponding separation unchanged. For example, if overprint mode is 1, then the operation `.2 .3 0 1 k` leaves the third (yellow) separation unchanged. It has a similar effect to `.2 .3 1 sc` in a `[/DeviceN [/Cyan /Magenta /Black] …]` color space. **PDF 1.3**

8.4.10 Black generation **PDF 1.2**

The black-generation function computes the value of the black component during conversion from **DeviceRGB** color space to **DeviceCMYK**. For additional information, see section 6.2.3, "Conversion from **DeviceRGB** to **DeviceCMYK**," of the *PostScript Language Reference Manual, Second Edition* [1]. The keyword for black generation is **BG**.

8.4.11 Undercolor removal

PDF 1.2

The undercolor removal function computes the amount to subtract from the cyan, magenta, and yellow components during conversion of color values from **DeviceRGB** color space to **DeviceCMYK**. See section 7.2.3, "Conversion from **DeviceRGB** to **DeviceCMYK**," of the *PostScript Language Reference Manual, Third Edition* [1]. The keyword for undercolor removal is **UCR**.

8.4.12 Transfer function

PDF 1.2

The transfer function adjusts the values of the gray color component. It is also a part of some halftone screens. For complete details, see section 6.3, "Transfer Functions," of the *PostScript Language Reference Manual, Second Edition* [1]. The keyword for transfer function is **TR**.

8.4.13 Halftone

PDF 1.2

The halftone parameter of the graphics state specifies how halftones should be produced. See 7.15, "Extended graphics states," for details about halftones. For general information on halftones, see section 6.4.3, "Halftone Dictionaries," of the *PostScript Language Reference Manual, Second Edition* [1]. The keyword for halftone is **HT**.

8.4.14 Halftone phase

PDF 1.2

The halftone phase parameters of the graphics state specifies the phase relationship of halftone cells to the coordinate axes. See section 7.3.3, "Halftone Phase," of the *PostScript Language Reference Manual, Second Edition* [1]. The keyword for halftone phase is **HTP**.

8.4.15 Smoothness

PDF 1.3

This parameter controls the quality of smooth shading (Type 2 patterns and the **sh** operator), and thus indirectly controls the rendering performance. Smoothness is the allowable color error between shading approximated with piecewise linear interpolation and the true shading of a possibly nonlinear shading function. The error is measured for each color component, and the maximum error is used. The allowable error (or tolerance) is specified as a percentage of the range of the color component. This percentage is expressed as a value from 0 to 1. Thus, a smoothness parameter of 0.1 represents a tolerance of 10% in each color component.

Each device may have internal limits on the maximum and minimum tolerances attainable. For example, setting smoothness to 1 may result in an internal smoothness of 0.5 on a high-quality color device, and setting smoothness to 0 on the same device may result in an internal smoothness of 0.01 if an error of that magnitude is imperceptible on that device.

The smoothness parameter may also interact with the accuracy of color conversion. In the case of a color conversion defined by a sampled function, the conversion function is unknown. Thus, the error may be sampled at too low a

frequency, in which case, the accuracy defined by the smoothness parameter cannot be guaranteed. In most cases, however, where the conversion function is smooth and continuous, the accuracy should be within the specified tolerance.

The effect of the smoothness parameter is similar to that of the flatness parameter. Note, however, that flatness is measured in device-dependent units of pixel width, whereas smoothness is measured as a percentage of color component range.

The keyword for smoothness is **SM**.

8.5 Color

8.5.1 Color parameters

8.5.1.1 Fill color

The fill color is used to paint the interior of paths and text characters that are filled. Filling is described in Section 8.6.2, "Path painting operators."

8.5.1.2 Stroke color

The stroke color is used to paint the border of paths and text that are stroked. Stroking is described in Section 8.6.2, "Path painting operators."

8.5.1.3 Fill color space

PDF 1.1

The fill color space is the color space in which the fill color is specified.

8.5.1.4 Stroke color space

PDF 1.1

The stroke color space is the color space in which the stroke color is specified.

8.5.1.5 Rendering intent

PDF 1.1

The rendering intent is a name that indicates the style of color rendering that should occur. See Section 7.13, "XObjects," and especially Table 7.36, "Color rendering intents," for further detail.

8.5.2 Color operators

PDF 1.1

The operators that set colors and color spaces fall into two classes. Operators in the first class, which were defined in PDF 1.0, set the color and color space at the same time, and they include only device-dependent color spaces. Operators in the second class, which are defined in PDF 1.1, set colors and color spaces separately, and they apply to all color spaces.

The default color space is **DeviceGray**, and the default fill and stroke colors are both black.

Color operators may appear inside Text Objects or at the Page-Description Level. See Figure 8.1.

8.5.2.1 Device-dependent color space operators

Arguments	*Operator*	*Semantics*
gray	**g**	Sets the color space to **DeviceGray** (or the **DefaultGray** color space, see Section 7.12.12 on page 245), and sets the gray tint to use for filling paths. *gray* is a number between 0 (black) and 1 (white). (**setgray** (fill))
gray	**G**	Sets the color space to **DeviceGray** (or the **DefaultGray** color space, see Section 7.12.12 on page 245), and sets the gray tint to use for stroking paths. *gray* is a number between 0 (black) and 1 (white). (**setgray** (stroke))
r g b	**rg**	Sets the color space to **DeviceRGB** (or the **DefaultRGB** color space, see Section 7.12.12 on page 245), and sets the color to use for filling paths. Each operand must be a number between 0 (minimum intensity) and 1 (maximum intensity). (**setrgbcolor** (fill))
r g b	**RG**	Sets the color space to **DeviceRGB** (or the **DefaultCMYK** color space, see Section 7.12.12 on page 245), and sets the color to use for stroking paths. Each operand must be a number between 0 (minimum intensity) and 1 (maximum intensity). (**setrgbcolor** (stroke))
c m y k	**k**	Sets the color space to **DeviceCMYK** (or the **DefaultCMYK** color space, see Section 7.12.12 on page 245), and sets the color to use for filling paths. Each operand must be a number between 0 (no ink) and 1 (maximum ink). The behavior of the **k** operator is affected by the overprint mode; see Section 8.4.9 on page 328. (**setcmykcolor** (fill))
c m y k	**K**	Sets the color space to **DeviceCMYK** (or the **DefaultRGB** color space, see Section 7.12.12 on page 245), and sets the color to use for stroking paths. Each operand must be a number between 0 (no ink) and 1 (maximum ink). The behavior of the **K** operator is affected by the overprint mode; see Section 8.4.9 on page 328. (**setcmykcolor** (stroke))

8.5.2.2 Generic color space operators

PDF 1.1

Arguments	*Operator*	*Semantics*
colorspace	**cs**	Sets the color space to use for filling paths. *colorspace* must be a name. If the color space is specified by a name (the device-dependent color spaces **DeviceGray**, **DeviceRGB**, and **DeviceCMYK**; or the **Pattern** color space for colored tiling patterns or shading patterns), then that name may be used. If it is specified by an array (all other color spaces), then *colorspace* must be a name defined in the current Resources dictionary. (**setcolorspace** (fill))

For example, the following expression is illegal:

```
[/CalGray dict] cs
```

Instead, one would write

```
/CS42 cs
```

and the Resources dictionary would contain

```
/CS42 [/CalGray dict]
```

The **cs** operator also sets the current fill-color to its initial value, which depends on the color space. For the device-dependent, calibrated, and **ICCBased** color spaces, the initial color is black. For a **Lab** color space, the initial value is specified by the minimum **Range** values. For an **Indexed** color space, the initial value is 0. The initial value in a **Separation** color space is 1, and the initial color value in a **Pattern** color space is a pattern that has an empty stream of marking operators, thus producing no marks on the page.

Arguments	*Operator*	*Semantics*
colorspace	**CS**	Same as **cs**, but for strokes. (**setcolorspace** (stroke))
c_1 c_2 c_3 c_4	**sc**	Sets the color to use for filling paths. The number of operands required and their interpretation is based on the current fill color space. For **DeviceGray**, **CalGray**, and **Indexed** color spaces, one operand is required. For **DeviceRGB**, **CalRGB**, and **Lab** color spaces, three operands are required. For **DeviceCMYK** and **CalCMYK**, four operands are required. (**setcolor** (fill))
c_1 c_2 c_3 c_4	**SC**	Same as **sc**, but for stroking paths. (**setcolor** (stroke))
c_1 … c_n	**scn**	PDF 1.2
c_1 … c_n *name*	**scn**	**scn** accepts the same parameters, and has the same effect, as **sc**. In addition, it supports **Pattern**, **Separation**, **ICCBased**, and **DeviceN** colors. (**setcolor** (fill for patterns))

If the current fill color space is a **Pattern** color space, then **scn** sets the pattern to use for filling paths. *name* is the name of a **Pattern** resource in the current Resources dictionary. If the pattern is uncolored (if **PatternType** is 1 and **PaintType** is 2), then the color is determined by the component values c_1 … c_n in the underlying color space. If the pattern is colored (if **PatternType** is 1 and **PaintType** is 1), or if it a shading pattern (if **PatternType** is 2), then the component values must not be specified.

If the current fill color space is a **Separation** color space, then **scn** sets the tint for filling paths to c_1, which is a number in the range 0 to 1 that represents the amount of colorant to be applied.

If the current fill color space is an **ICCBased** color space, then **scn** sets the color values for filling paths to c_1 … c_n, which are numbers in the range 0 to 1. PDF 1.3

If the current fill color space is a **DeviceN** color space, then **scn** sets the tints for filling paths to c_1 … c_n, which are numbers in the range 0 to 1 that represents the amount of each colorant to be applied. (**setcolor** (fill)) PDF 1.3

c_1 … c_n	**SCN**	PDF 1.2
c_1 … c_n *name*	**SCN**	Same as **scn**, but for strokes. (**setcolor** (stroke))

8.5.2.3 Color rendering intent PDF 1.1

Arguments	*Operator*	*Semantics*
intent	**ri**	Sets the color rendering intent in the graphics state.

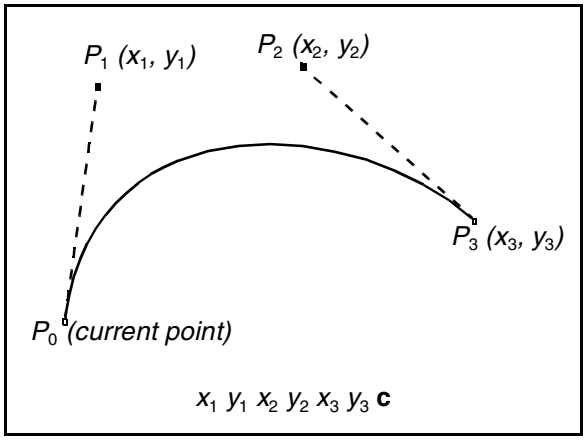
intent is a name of a color rendering intent, which indicates the style of color rendering that should occur, as described in Table 7.36 on page 252.

8.6 Paths

Paths are used to represent lines, curves, and regions. A path consists of a series of path segment operators describing where marks are to appear on the page, followed by a path painting operator, which actually marks the path in one of several ways. A path may be composed of one or more disconnected sections, referred to as *subpaths*. An example of a path with two subpaths is a path containing two parallel line segments.

Path segments may be straight lines or curves. Curves in PDF files are represented as cubic Bézier curves. A cubic Bézier curve is specified by the *x*- and *y*-coordinates of four points: the two endpoints of the curve (the current point, P_0, and the final point, P_3) and two *control points* (points P_1 and P_2), as shown in Figure 8.7.

Figure 8.7 *Bézier curve*



Once these four points are specified, the cubic Bézier curve R(t) is generated by varying the parameter t from 0 to 1 in the following equation:

$$R(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3$$

In this equation, P_0 is the current point before the curve is drawn. When the parameter t has the value 0, $R(t) = P_0$ (the current point). When $t = 1$, $R(t) = P_3$. The curve does not, in general, pass through the two control points P_1 and P_2.

Bézier curves have two desirable properties. First, the curve is contained within the convex hull of the control points. The convex hull is most easily visualized as the polygon obtained by stretching a rubber band around the outside of the four points defining the curve. This property allows rapid testing of whether the curve is completely outside the visible region, and so does not have to be rendered. Second, Bézier curves can be very quickly split into smaller pieces for rapid rendering.

Note *In the remainder of this book, the term* Bézier curve *means cubic Bézier curve.*

Paths are subject to and may also be used for clipping. Path clipping operators replace the current clipping path with the intersection of the current clipping path and the current path.

<path> ::= <subpath>+
[path clipping operator]
<path painting operator>

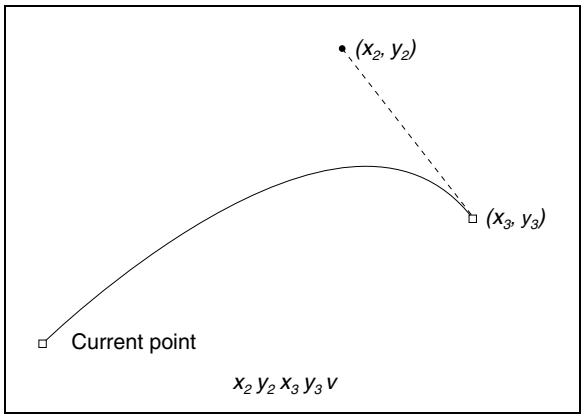
<subpath> ::= **m** <path segment operator except **m** and **re**>* |
re

8.6.1 Path segment operators

All operands are numbers that are coordinates in user space.

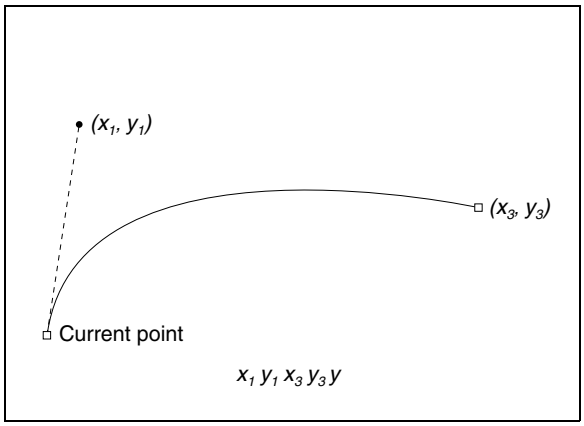
Arguments	*Operator*	*Semantics*
x y	**m**	Moves the current point to *(x, y)*, omitting any connecting line segment. (**moveto**)
x y	**l**	(operator is lowercase L) Appends a straight line segment from the current point to *(x, y)*. The new current point is *(x, y)*. (**lineto**)
$x_1\ y_1\ x_2\ y_2\ x_3\ y_3$	**c**	Appends a Bézier curve to the path. The curve extends from the current point to (x_3, y_3) using (x_1, y_1) and (x_2, y_2) as the Bézier control points, as shown in Figure 8.7. The new current point is (x_3, y_3). (**curveto**)
$x_2\ y_2\ x_3\ y_3$	**v**	Appends a Bézier curve to the current path between the current point and the point (x_3, y_3) using the current point and (x_2, y_2) as the Bézier control points, as shown in Figure 8.8. The new current point is (x_3, y_3). (**curveto** (first control point coincides with initial point on curve))

Figure 8.8 *v operator*



$x_1\ y_1\ x_3\ y_3$	**y**	Appends a Bézier curve to the current path between the current point and the point (x_3, y_3) using (x_1, y_1) and (x_3, y_3) as the Bézier control points, as shown in Figure 8.9. The new current point is (x_3, y_3). (**curveto** (second control point coincides with final point on curve))

Figure 8.9 y *operator*



x y width height **re** Adds a rectangle to the current path.

width and *height* are distances in user space. The operation

```
x  y  width  height  re
```

is defined to have the same effect as the sequence

```
x  y  m
x+width  y  l
x+width  y+height  l
x  y+height  l
h
```

h Closes the current subpath by appending a straight line segment from the current point to the starting point of the subpath. (**closepath**)

8.6.2 Path painting operators

Paths may be stroked and/or filled. As in the PostScript language, painting completely obscures any marks already on the page under the region that is painted.

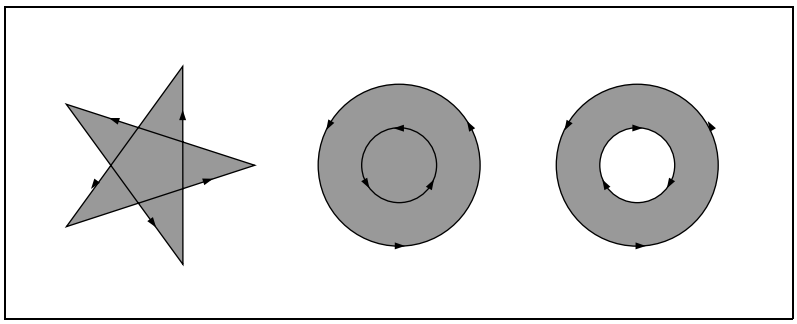
Stroking draws a line along the path, using the line width, dash pattern, miter limit, line cap style, line join style, stroke color, stroke color space, and stroke adjustment from the graphics state. The line drawn when a path is stroked is centered on the path. If a path consists of multiple subpaths, each is treated separately.

The process of filling a path paints the entire region enclosed by the path, using the fill color and fill color space. If a path consists of several disconnected subpaths, each is filled separately. Any open subpaths are implicitly closed before being filled. Closing is accomplished by adding a segment between the first and last

points on the path. For a simple path, it is clear what lies inside the path and should be painted by a fill. For more complicated paths, it is not so obvious. One of two rules is used to determine which points lie inside a path.

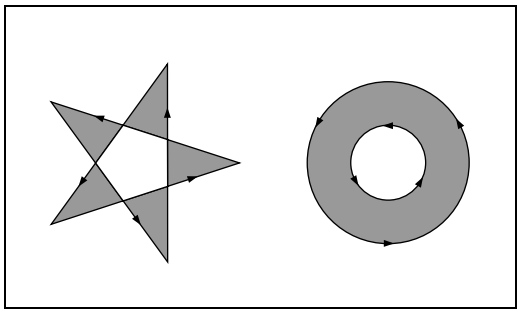
The *non-zero winding number rule* uses the following test to determine whether a given point is inside a path and should be painted. Conceptually, a ray is drawn in any direction from the point in question to infinity, and the points where the ray crosses path segments are examined. Starting from a count of zero, add one to the count each time a path segment crosses the ray from left to right, and subtract one from the count each time a path segment crosses the ray from right to left. If the ray encounters a path segment that coincides with it, the result is undefined. In this case, a ray in another direction can be picked, since all rays are equivalent. After counting all the crossings, if the result is zero then the point is outside the path. The effect of using this rule on various paths is illustrated in Figure 8.10. The non-zero winding number rule is used by the PostScript language **fill** operator.

Figure 8.10 *Non-zero winding number rule*



The *even–odd rule* uses a slightly different strategy. The same calculation is made as for the non-zero winding number rule, but instead of testing for a result of zero, a test is made as to whether the result is even or odd. If the result is odd, the point is inside the path; if the result is even, the point is outside. The result of applying this rule to various paths is illustrated in Figure 8.11. The even–odd rule is used by the PostScript language **eofill** operator.

Figure 8.11 *Even–odd rule*



Arguments	*Operator*	*Semantics*
	n	Ends the path without filling or stroking it. This is a "path painting no-op," primarily used with a path clipping operator (see Section 8.6.3, "Path clipping operators"), but like the other path painting operators, it terminates a Path Object. (**newpath**)
	S	Strokes the path. (**stroke**)
	s	Similar to the **S** operator, but closes the path before stroking it. (**closepath** and **stroke**) `s` has the same effect as `h S`.
	f	Fills the path, using the non-zero winding number rule to determine the region to fill. (**fill**)
	F	Same as the **f** operator. Included only for compatibility. Although applications that read PDF files must be able to accept this operator, applications that generate PDF files should use the **f** operator instead. (**fill**)
	f*	Fills the path, using the even–odd rule to determine the region to fill. (**eofill**)
	B	**fill** and **stroke**. `B` has the same effect as `q f Q S`.
	b	**closepath**, **fill**, and **stroke**. `b` has the same effect as `h B`.
	B*	**eofill** and **stroke**. `B*` has the same effect as `q f* Q S`.
	b*	**closepath**, **eofill**, and **stroke**. `b*` has the same effect as `h B*`.
name	**sh**	When a path that is to be filled with a gradient (see Section 7.17 on page 287) has the same geometry as the gradient itself, it is not necessary to define the gradient as a Type 2 pattern, define the path separately, and then use the **f** (fill) operator. Instead, one may use the **sh** operator to paint the same area. (**shfill**) **PDF 1.3** *name* is the name of a **Shading** dictionary in the current Resources dictionary. (If its **ShadingType** is greater than 3, this dictionary is part of a stream.) All coordinates in the Shading dictionary are interpreted relative to the current user space. (When a Shading dictionary is used in a pattern, the coordinates are expressed in pattern space.)

8.6.3 Path clipping operators

Path clipping operators cause the current clipping path to be replaced with the intersection of the current clipping path and the path. A path is made into a clipping path by inserting a path clipping operator (**W** or **W***) between the last path segment operator and the path painting operator.

Although the path clipping operator appears before the path painting operator, the path clipping operator does not alter the clipping path at the point it appears. Rather, it modifies the effect of the path painting operator. After the path is filled,

stroked, or ended by the path painting operator, it is set to be the current clipping path. If the path is both filled and stroked, the painting is done in that order before making the path the current clipping path.

The definition of the clipping path and all subsequent operations it is to affect should be contained between a pair of **q** and **Q** operators. Execution of the **Q** operator causes the clipping path to revert to that saved by the **q** operator, before the clipping path was modified.

Arguments	*Operator*	*Semantics*
	W	Uses the non-zero winding number rule to determine which regions are inside the clipping path. (**clip**)
	W*	Uses the even–odd rule to determine which regions are inside the clipping path. (**eoclip**)

8.7 Text state

The text state is composed of those graphics state parameters that affect only text.

8.7.1 Text State parameters and operators

There are nine parameters in the text state, each of which can be set individually:

1. T_c is the character spacing parameter.
2. T_w is the word spacing parameter.
3. T_h is the horizontal spacing parameter.
4. T_l is the “leading” parameter.
5. T_f is the text font.
6. T_{fs} is the text font size.
7. T_m is the text matrix.
8. T_{mode} is the rendering mode.
9. T_{rise} is the “text rise”.

There are two additional parameters of the text state:

1. T_{LM} is the matrix for the current text line.
2. T_{RM} is the rendering matrix.

Each of the items in the text state is described in the following sections.

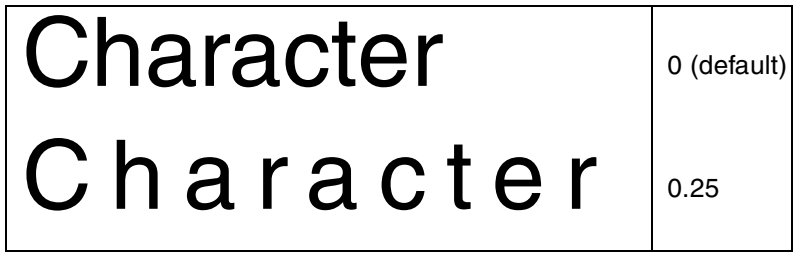
Note Section 8.7.4, "Text rendering," *describes how these parameters are used, and their exact effects on the text state.*

Note *These operators can appear outside of text objects, and the values they set are retained across text objects on a single page. Like other graphics state parameters, the values are initialized to the default values at the beginning of each page.*

8.7.1.1 Character spacing

The character spacing parameter, T_c, is a number specified in text space units. It is *added* to the displacement between the origin of one character and the origin of the next. See Figure 7.3 on page 209 for examples of character origins and displacements. In the default coordinate system, the positive direction of the *x*-axis points to the right, and the positive direction of the *y*-axis points upward. So for horizontal writing, a positive value of T_c has the effect of expanding the space between characters; see Figure 8.12. For vertical writing, however, a *negative* value of T_c has the effect of expanding the space between characters.

Figure 8.12 *Character spacing for horizontal writing*



Character spacing is applied to each glyph in the string, regardless of the number of bytes used for that glyph's character code. Therefore character spacing is used even with fonts that have multi-byte encodings.

Arguments	*Operator*	*Semantics*
charSpace	**Tc**	Set character spacing Sets T_c to *charSpace*. Character spacing is used, together with word spacing, by the **Tj**, **TJ**, and **'** operators. *charSpace* is a number expressed in text space units and has an initial value of 0.

8.7.1.2 Word spacing

The word spacing parameter, T_w, is a number specified in text space units. It works in the same way as character spacing, but applies only to the space character, <20>. T_w is added to the displacement between the origin of the space character and the origin of the following character. For horizontal writing, a positive value for T_w has the effect of increasing the spacing between words. For vertical writing, a positive

value for T_w *decreases* the space between words, since the positive direction of the *y*-axis points upward; therefore a negative value will increase the space between words. Figure 8.13 illustrates the effect of word spacing in horizontal writing.

Figure 8.13 *Effect of word spacing in horizontal writing*

Word Space	0 (default)
Word Space	2.5

Word spacing is applied to every instance of the single byte <20> in a string. Therefore word spacing is not used with fonts that have only multi-byte encodings or with fonts whose encodings do not use the single byte <20> as the space character.

Arguments	*Operator*	*Semantics*
wordSpace	**Tw**	Set word spacing Sets T_w to *wordSpace*. Word spacing is used by the **Tj**, **TJ**, and **'** operators. *wordSpace* is a number expressed in text space units and has an initial value of 0.

8.7.1.3 Horizontal scaling

The horizontal scaling parameter, T_h, adjusts the width of characters by stretching or shrinking them in the horizontal direction. The scaling is specified as a percent of the normal width of the characters, with 100 being the normal width. Figure 8.14 shows the effect of horizontal scaling. The scaling always applies to the *x* coordinate, independent of the writing mode.

Figure 8.14 *Horizontal scaling*

Word	100 (default)
WordWord	50

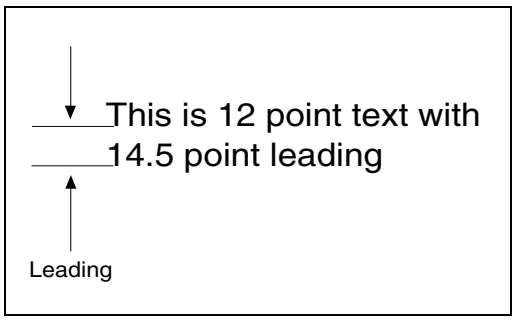
Arguments	*Operator*	*Semantics*
scale	**Tz**	Set horizontal scaling

Sets T_h to (*scale* ÷ 100). *scale* is a number expressed in percent of the normal scaling and has an initial value of 100.

8.7.1.4 Leading

The leading parameter, T_l, is measured in text space units. It specifies the vertical distance between the baselines of adjacent lines of text, as shown in Figure 8.15. The leading parameter is used by the **TD**, **T***, ', and " operators; it is independent of the writing mode.

Figure 8.15 *Leading*



Arguments	*Operator*	*Semantics*
leading	**TL**	Set text leading
		Sets T_l to *leading*. The **TL** operator need not be used in a PDF file unless the **T***, ', or " operators are used. *leading* has an initial value of 0.

8.7.1.5 Text font and size

Arguments	*Operator*	*Semantics*
fontname size	**Tf**	Set font and size
		Sets T_f to *fontname* and T_{fs} to *size*. There is no initial value for either *fontname* or *size*; they must be specified using **Tf** before drawing any text. *fontname* is the name of a Font in the current Resources dictionary. *size* is a number expressed in text space units.

8.7.1.6 Text matrix

The text matrix specifies the transformation from text space (see Section 3.3, "Text space") to user space. The text matrix is set with the **Tm** operator (see page 345).

8.7.1.7 Text rendering mode

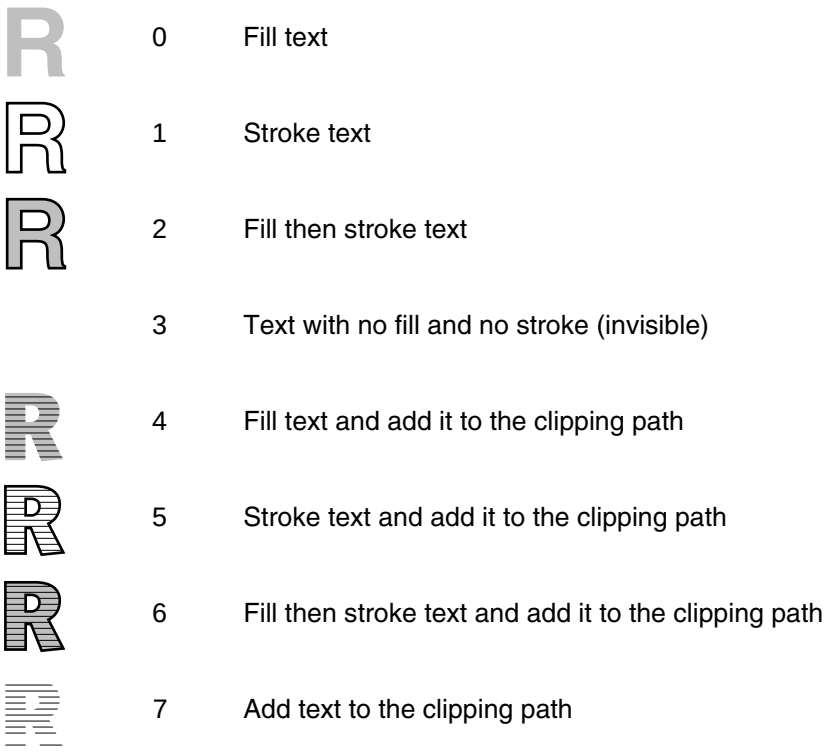
Determines whether text is stroked, filled, or used as a clipping path.

Note *The text rendering mode has no effect on text displayed using a Type 3 font.*

The rendering modes are shown in Figure 8.16. In the figure, a stroke color of black and a fill color of light gray are used. After one of the clipping modes is used for text rendering, the text object must be ended using the **ET** operator before changing the text rendering mode.

Note *For the clipping modes (4–7), a series of lines has been drawn through the characters in Figure 8.16 to show where the clipping occurs.*

Figure 8.16 *Text rendering modes*



Rendering mode	Description
0	Fill text
1	Stroke text
2	Fill then stroke text
3	Text with no fill and no stroke (invisible)
4	Fill text and add it to the clipping path
5	Stroke text and add it to the clipping path
6	Fill then stroke text and add it to the clipping path
7	Add text to the clipping path

Arguments	*Operator*	*Semantics*
render	**Tr**	Set the text rendering mode.

Sets T_{mode} to *render*, which is an integer and has an initial value of 0.

8.7.1.8 Text rise

Text rise specifies the amount, in text space units, to move the baseline up or down from its default location. Positive values of text rise move the baseline up. Adjustments to the baseline are useful for drawing superscripts or subscripts. The default location of the baseline can be restored by setting the text rise to 0. Figure 8.17 illustrates the effect of the text rise, which is set using the **Ts** operator. Text rise always applies to the *y* coordinate, regardless of the writing mode.

Figure 8.17 *Text rise*

This text is superscripted	(This text is) Tj 5 Ts (superscripted) Tj
This text is subscripted	(This text is) Tj –5 Ts (subscripted) Tj
This text moves around	(This) Tj –5 Ts (text) Tj 5 Ts (moves) Tj 0 Ts (around) Tj

Arguments	*Operator*	*Semantics*
rise	**Ts**	Set text rise. Sets T_{rise} to *rise*, which is a number expressed in text space units and has an initial value of 0.

8.7.2 Text Object operators

A PDF text object consists of operators that specify character strings, movement of the current point, and text state. A text object begins with the **BT** operator and ends with the **ET** operator. See Figure 8.1 on page 320.

<text object> ::= **BT**
<text operator or graphics state operator>*
ET

When **BT** is encountered, the text matrix is initialized to the identity matrix. When **ET** is encountered, the text matrix is discarded. Text objects cannot be nested—a second **BT** cannot appear before an **ET**.

Note *If a page does not contain any text, no text operators (including operators that merely set the text state) may be present in the page description.*

Arguments	*Operator*	*Semantics*
	BT	Begins a Text Object. Initializes the text matrix, T_m, and the line matrix, T_{LM}, to the identity matrix.
	ET	Ends a Text Object. Discards the text matrix.

8.7.3 Text positioning operators

A text object keeps track of the current point and the start of the current line. The text string operators move the current point as the various forms of the PostScript language **show** operator do. Operators that move the start of the current line move the current point as well.

Note *These operators may appear only within text objects.* See Figure 8.1 on page 320.

Arguments	*Operator*	*Semantics*
t_x t_y	**Td**	Moves to the start of the next line, offset from the start of the current line by (t_x, t_y). t_x and t_y are numbers expressed in text space units. More precisely, **Td** performs the following assignments: $$T_m = T_{LM} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} \times T_{LM}$$
t_x t_y	**TD**	Moves to the start of the next line, offset from the start of the current line by (t_x, t_y). As a side effect, this sets the leading parameter in the text state. t_x t_y `TD` is defined to have the same effect as `-`t_y `TL` t_x t_y `Td`
a b c d x y	**Tm**	Sets the text matrix, T_m, and the text line matrix, T_{LM}. It also sets the current point and line start position to the origin. **Tm** performs the following assignments: $$T_m = T_{LM} = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ x & y & 1 \end{bmatrix}$$ The operands are all numbers, and the initial value for T_m and T_{LM} is the identity matrix, `[1 0 0 1 0 0]`. Although the operands specify a matrix, they are passed to **Tm** as six numbers, not as an array.

The matrix specified by the operands passed to the **Tm** operator is not concatenated onto the current text matrix, but replaces it.

T* Moves to the start of the next line.

`T*` is defined to have the same effect as `0` T_l `Td`

where T_l is the leading parameter of the text state.

8.7.4 Text rendering

Before text is rendered by the **Tj** or **TJ** operator, it is placed and transformed according to the parameters in the text state. The *rendering matrix* for the text is computed as follows:

The current text matrix, T_m, is translated by the text rise, T_{rise}. Next, that is scaled by the font size, T_{fs}, and the horizontal text scale, T_h. Finally, that is concatenated to the current transformation matrix in the graphics state (*CTM*) to produce the rendering matrix, T_{RM}:

$$T_{RM} = \begin{bmatrix} T_{fs} \times T_h & 0 & 0 \\ 0 & T_{fs} & 0 \\ 0 & T_{rise} & 1 \end{bmatrix} \times T_m \times CTM$$

This calculation occurs, in effect, whenever any of the text parameters change, before **Tj** or **TJ** occur. When text is rendered, the text line matrix, T_{LM}, is unaffected, but the text matrix, T_m, is translated by the origin-displacement of the text, which affects subsequent rendering operations, as shown above. For horizontal-mode writing, the origin-displacement is along the *x* axis; for vertical writing (see Section 7.7.8 on page 208), the displacement is along the *y* axis.

8.7.5 Text string operators

These operators draw text on the page. Although it is possible to pass individual characters to the text string operators, text searching performs significantly better if the text is grouped by word and paragraph.

PDF supports the same conventions as the PostScript language for specifying non-printable ASCII characters. That is, a character can be represented by an escape sequence, as described in Table 4.1 on page 38.

Note *The default current point is at the page origin. Therefore, unless some prior operation in the same text object changes the current point, the text will appear at the origin. It is suggested that a* **Tm** *operation be used to establish the initial current point in a text object at the position in text space where initial text is to appear. Subsequent text operations may change the current point.*

Arguments	*Operator*	*Semantics*
string	**Tj**	Shows text string, using the character and word spacing parameters from the text state. (**show**)
string	**'**	Moves to next line and shows text string, using the character and word spacing parameters from the text state. (**show**) *string* **'** is defined to have the same effect as **T*** *string* **Tj**
a_w a_c *string*	**"**	Moves to next line and shows text string. a_w and a_c are numbers expressed in text space units. a_w specifies the additional space width, and a_c specifies the additional space between characters. (**show**) a_w a_c *string* **"** is defined to have the same effect as a_w **Tw** a_c **Tc** *string* **'**
	Note	*The values specified by* a_w *and* a_c *remain the word and character spacings after the* **"** *operator is executed.*
[number or string …]	**TJ**	Shows text string, allowing individual character positioning, and using the character and word spacing parameters from the text state. (**show** with displacements)

For each element of the array, if the element is a string, **TJ** shows the string. If it is a number, it is expressed in thousandths of an em. (An *em* is a typographic unit of measurement equal to the size of a font. For example, in a 12-point font, an em is 12 points.) **TJ** *subtracts* this amount from the current *x* coordinate in horizontal writing mode, or from the current *y* coordinate in vertical writing mode. In the normal case of horizontal writing in the default coordinate system, this has the effect of moving the current point to the *left* by the given amount.

Each character is first justified according to any character and word spacing settings made with the **Tc**, **Tw**, or **"** operators, and then any numeric offset present in the array passed to the **TJ** operator is applied. An example of the use of **TJ** is shown in Figure 8.18.

Figure 8.18 *Operation of* **TJ** *operator in horizontal writing*

AWAY again	[(AWAY again)] TJ
AWAY again	[(A) 120 (W) 120 (A) 95 (Y again)] TJ

8.7.6 Text strings in multi-byte fonts

PDF 1.2

The text string operators can be used with any string. For strings that use multi-byte encodings, the high-order byte of a character code must appear first. The strings must conform to the syntax for string objects. Therefore care must be taken when including multi-byte character codes. These codes may contain single-byte values that are the same as the ASCII characters for left parenthesis (<28>), right parenthesis (<29>), and backslash (<5C>). When a string is written by enclosing the data in parentheses, these bytes must be preceded by the backslash character. All other byte values between <00> and <FF> may be used in a string object.

8.8 External objects (XObjects)

PDF defines three types of XObjects: Image XObjects, Form XObjects, and PostScript XObjects.

8.8.1 XObject operators

The **Do** operator permits the execution of an arbitrary object whose data is encapsulated within a PDF object. The currently defined XObjects are images and PostScript language forms, discussed in Section 7.13, "XObjects."

Arguments	*Operator*	*Semantics*
xobject	**Do**	Executes the specified XObject. *xobject* must be the name of an Image, Form, or PostScript XObject in the current Resources dictionary. See Section 7.13, "XObjects.
string	**PS**	The **PS** operator provides an in-line equivalent to a PostScript XObject. The **PS** operator has one argument, a string. When a **PS** operator is encountered while a document is being printed to a PostScript printer, the contents of the string are placed into the PostScript output as the argument of an instance of the PostScript operator **exec**. This string is copied without interpretation and may include PostScript comments. In any other case, the **PS** operator has no other effect. See Section 7.13.8 on page 257 for additional information.

PDF 1.1

8.9 In-line image objects

In addition to the Image XObject described in Section 7.13, "XObjects," PDF supports in-line images. An in-line Image Object consists of the operator **BI**, followed by Image XObject key–value pairs, followed by the operator **ID**, followed by the image data, followed by **EI**:

<in-line image> ::=
BI
<Image XObject key–value pairs>

ID
<lines of data>*
EI

Note *If an in-line image does not use* **ASCIIHexDecode** *or* **ASCII85Decode** *as one of its filters,* **ID** *should be followed by a single space. The character following the space is interpreted as the first byte of image data.*

Image data may be encoded using any of the standard PDF filters. The key–value pairs provided in an in-line image should not include keys specific to resources: **Type**, **Subtype**, and **Name**. Within in-line images, the standard key names may be replaced by the shorter names listed in Table 8.1. These abbreviations may not be used in Image XObjects, however.

Table 8.1 *Abbreviations for in-line image names*

Name	*Abbreviated name*	
ASCIIHexDecode	**AHx**	
ASCII85Decode	**A85**	
BitsPerComponent	**BPC**	
CCITTFaxDecode	**CCF**	
ColorSpace	**CS**	
DCTDecode	**DCT**	
Decode	**D**	
DecodeParms	**DP**	
DeviceCMYK	**CMYK**	
DeviceGray	**G**	
DeviceRGB	**RGB**	
Filter	**F**	
FlateDecode	**Fl**	PDF 1.2
Height	**H**	
ImageMask	**IM**	
Indexed	**I**	
Intent	*no abbreviation*	PDF 1.1
Interpolate	**I**	
LZWDecode	**LZW**	
RunLengthDecode	**RL**	
Width	**W**	

Note *The in-line format should be used only for small images (4K or less) because viewer applications have less flexibility when managing in-line image data.*

In-line images, like Image XObjects, are one unit wide and one unit high in user space and drawn at the origin. Images are sized and positioned by transforming user space using the **cm** operator.

Arguments	*Operator*	*Semantics*
	BI	Begins image
	ID	Begins image data
	EI	Ends image

The value of the **CS** or **ColorSpace** key may be a device-dependent color space (**DeviceGray**, **DeviceRGB**, or **DeviceCMYK**, or its abbreviation from the preceding table). The value may not be a device-independent color space or a special color space, with the exception of a limited form of the **Indexed** color space, which may be written as

```
[/Indexed base hival lookup]
```

where *base* is a device-dependent color space and *lookup* is a string; see Section 7.12.10, "Indexed color spaces." The name **/Indexed** may be abbreviated as **/I**.

In PDF 1.2, the value may also be the name of a color space in the current Resources dictionary. In this case, any color space that may be used with an Image XObject may be used for the in-line image (see Section 7.13.1, "Images").

Example 8.1 shows a 17×17 sample in-line image. The image has 8 bits per component; it is an RGB image that has been LZW and ASCII85 encoded. The **cm** operator has been used to scale the image to render at a size of 17×17 user space units and to be located at an *x*-coordinate of 298 and a *y*-coordinate of 388. The **q** and **Q** operators limit the scope of the **cm** operator's effect to resizing the image.

Example 8.1 *In-line image*

```
q
17 0 0 17 298 388 cm
BI
/W 17
/H 17
/BPC 8
/CS /RGB
/F [/A85 /LZW]
ID
J1/gKA>.]AN&J?]-<HW]aRVcg*bb.\eKAdVV%/PcZ
... omitted data ...
```

```
R.s(4KE3&d&7hb*7[%Ct2HCqC~>
EI
Q
```

8.10 Other operators

8.10.1 Type 3 font operators

Type 3 font operators can be used only within the character definitions inside a Type 3 font. Each Type 3 font definition must begin with either a **d0** or **d1** operator. See Section 5.7 of the *PostScript Language Reference Manual, Third Edition* [1] for details.

Arguments	*Operator*	*Semantics*
w_x w_y	**d0**	(d zero) **setcharwidth** The operands are both numbers.
w_x w_y ll_x ll_y ur_x ur_y	**d1**	(d one) **setcachedevice** The operands are all numbers.

8.10.2 Compatibility operators

PDF 1.1

PDF does not specify a viewer's behavior when it encounters an undefined page description operator. However, Appendix G does describe the behavior of the Adobe Acrobat viewers. An Acrobat viewer usually alerts the user when it encounters an undefined page description operator. The operators below modify this behavior.

Arguments	*Operator*	*Semantics*
	BX	This operator directs a viewer to not report any undefined operators until a matching **EX** is encountered. (**BX**–**EX** pairs may nest.)
	EX	This operator ends a section of page description in which undefined operators should not be reported.

8.10.3 Marked Content operators

PDF 1.2

The Marked Content operators are used in page descriptions such as the **Contents** stream of a page to indicate a part of the stream that may be significant to an application other than a strict PDF consumer, such as a PDF Viewer. The content

that is marked is not a sequence of bytes in the stream, but a sequence of graphics objects. Each graphics object is fully qualified by the graphics state in which it is rendered.

For example, a graphics application might use these operators to indicate that a certain set of objects constitute a "group." A text-processing application might use them to maintain a connection between a footnote number in the running text and the footnote itself at the bottom of the page.

There are two kinds of marks, those that bracket a sequence of objects, and those that mark a place in the stream. Bracketed sequences begin with either **BMC** or **BDC**, and they end with **EMC**. **BDC** has the same effect as **BMC** but includes a property list as additional information. Places are marked with either **MP** or **DP**. **DP** has the same effect as **MP** but, like **BDC**, includes a property list.

These operators may appear only *between* graphics objects; they may not occur within a graphics object nor between a graphics state operator and its operands. See Figure 8.1 on page 320.

Bracketed sequences may be nested within each other. A bracketed sequence must be entirely contained within a single stream; it may not cross page boundaries, for example. (The **Contents** key of a Page object is permitted to be either a stream or an array of streams; such an array is considered to be a single stream.)

When Marked Content is used with text, the begin-end Marked Content operators (**BMC**/**BDC** and **EMC**) and the begin-end text operators (**BT** and **ET**) must be properly (separately) nested. That is, the sequence `BMC BT … EMC ET` is illegal, as is `BT BMC … ET EMC`. The sequence `BMC BT … ET EMC` is legal, as is the sequence `BT BMC … EMC ET`.

The **BMC** and **MP** operators have only one operand, a *tag* which indicates the role of the operator. The **BDC** and **DP** operators have an additional operand, a list of *properties* that are associated with the mark and whose interpretation is relative to the tag. The properties are represented by a dictionary. This dictionary may be written inline in the content stream if all its values are direct objects. If any value is an indirect object (referring to an object outside the stream), then the list is specified by the name of a Property List in the current Resources dictionary. (See page 313.)

With the exception of the **Subtype** key, PDF makes no assumptions about the properties; interpretation of this dictionary is up to the application or PDF extension that placed the content markers in the stream. It is suggested, however, that any particular extension use keys in a consistent way and always use the same type (or small set of types) for the values of a particular key.

The *tags* that are associated with marks must be registered (see Appendix F) to prevent conflicting usage when more than one application may be marking a particular content stream. The components of the name, including the registered prefix, must be separated by a single period, and the tag may not begin with a period.

Arguments	*Operator*	*Semantics*
tag	**BMC**	Begin marked content. **BMC** indicates the beginning of a sequence of graphics objects that is "marked" in some way. *tag* must be a name; it should indicate the role of the content that is marked.
tag properties	**BDC**	Begin marked content with a property list. **BDC** indicates the beginning of a sequence of graphics objects that is "marked" in some way. *tag* must be a name; it should indicate the role of the content that is marked. *properties* is either an inline dictionary, that is, a direct object dictionary in the content stream, or it is the name of a property list in the current Resources dictionary.
	EMC	End marked content. **EMC** indicates the end of a marked sequence of graphics objects. Sequences may be nested.
tag	**MP**	Mark a point in the content. **MP** indicates a place within the sequence of graphics objects that is "marked." **MP** is not intended for use when some subsequence of the content is being marked: **BMC** and **EMC** should be used when the beginning and end of a subsequence is to be indicated. *tag* must be a name; it should indicate the role of the place that is marked.
tag properties	**DP**	Mark a point in the content and include a property list. **DP** is similar to **MP**, but includes a property list, as **BDC** does.

Marked content and clipping

When Marked Content is used to bracket a path or text clip object, then additional restrictions apply. Apath object may or may not include a clip, and it may or may not be painted; the same is true of text objects. A "clip object" is either an *unpainted, clipped path object* (defined by a sequence including a path segment operator, a clip operator, followed by the **n** operator) or an *unpainted, clipped text object* (defined by a sequence in which text is painted in text-rendering mode 7). If a Marked Content includes *only* clip objects, then the Marked Content applies to those objects. Otherwise, Marked Content does not apply to clip objects.

Nesting of clip objects within Marked Content is allowed. For example, if multiple lines of text are used to mask an image, each line of text may be bracketed by Marked Content, and, the lines of bracketed text may be bracketed by an outer Marked Content. An empty Marked Content within a clip Marked Content is considered to be a nested within the clip Marked Content. An additional restriction is that the save and restore operators (**q** and **Q**) may not occur within Marked Content that is used to bracket clip objects.

The precise rules for determining whether a Marked Content applies to a clip object are as follows:

1. If the only objects within a Marked Content are clip objects, then the Marked Content applies to those clip objects.

2. A Marked Content that contains only clip objects is a clip object.

3. An empty Marked Content that is contained by a clip Marked Content is part of the clip Marked Content. A Marked Place (denoted by **MP** and **DP**) is treated the same as an empty Marked Content.

4. If both clip and marking objects occur between Marked Content delimiters, then the clip objects are not marked by the enclosing Marked Content. That is, any Marked Content attributes do not apply to the clip objects.

5. The largest tree of nested Marked Content operators that contains only empty Marked Content and clip Marked Content is a clip Marked Content.

6. The save and restore operators (**q** and **Q**) may not occur within a Marked Content that is used to bracket clip objects.

7. Marked Content must nest within **BT**/**ET**, and **BT**/**ET** must nest within Marked Content.

8. Invisible graphic objects inside Marked Content are treated as rendered objects. They are not clip objects.

Examples

Example 1:

```
/Clip BMC
    100 100 10 10 re W n                clip path
    (Clip me) Tj                object that is clipped
EMC
```

Example 2:

```
/Clip BMC
    /PointText <<...>> BDC
    BT
        7 Tr                        begin text clip mode
        /Pgf BMC
            (Line 1) Tj
        EMC
        /Pgf BMC
            (Line) Tj ( 2) Tj
        EMC
    ET                              set current text clip
    EMC
    100 100 10 10 re f                  filled path
EMC
```

Example 3:

```
/G1 BMC
    /G2 BMC
```

```
        /G3 BMC
            0 0 m
            100 100 l
            0 100 1 W n                 clip path 1
            0 0 m
            200 200 l
            0 100 1 f                   filled path
        EMC
        /G4 BMC
            0 0 m
            300 300 l
            0 100 1 W n                 clip path 2
        EMC
    EMC
    100 100 10 10 re f                  filled path
EMC
```

Example 3 shows how nested clip Marked Content is handled. **G3** does not apply to clip path 1 because **G3** also includes a filled path. **G4** *does* apply to clip path 2, but **G2** does *not* apply to clip path 2.

Example 4:

```
/1 BMC
    <clip path>
    /2 BMC
        /3 BMC
        EMC
    DP
    EMC
EMC
```

BMC 2 contains only an empty Marked Content, and a DP. However, they are all considered clip objects because they are all nested in BMC 1 which is clip-only and has a real clip object in it. BMC 2 is not empty, but it only contains empty Marked Content, so it is a clip object. The same rule applies to BMC 3.

Example 5:

```
/1 BMC
    /2 BMC
        /3 BMC
        EMC
    EMC

    /4 BMC
        <clip path>
    EMC
EMC
```

Here, BMC 1 becomes clip-only due to the nested clip path, and BMC 2 and MC 3 become clip objects due to their containment in BMC 1.